305233

UNLIMITED

(2)

AD-A242 148

Report No. 91024

Report No. 91024

ROYAL SIGNALS AND RADAR ESTABLISHMENT, MALVERN

A FORMAL DEFINITION OF THE STATIC SEMANTICS OF ELLA'S CORE

Authors: J D Morison & M G Hill

PROCUREMENT EXECUTIVE, MINISTRY OF DEFENCE
RSRE
Malvern, Worcestershire.

August 1991

UNLIMITED

CONDITIONS OF RELEASE

0110617 305233

************************* DRIC U

************************* DRIC Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

Royal Signals and Radar Establishment

Report 91024

Title: A FORMAL DEFINITION OF THE STATIC SEMANTICS OF ELLA'S CORE

Authors: J D Morison, M G Hill

Date: August 1991

Summary

At the heart of the full ELLA language are a set of Core constructs into which any ELLA description can be transformed. This document describes a set of formal transformation rules which map these Core constructs into a set of data structures. These transformation rules define the static semantics of the language. Examples are given of circuits which are translated from the full language into ELLA's Core and of Core circuits which are translated via the formal transformation system into a set of data structures.

INTENTIONALLY BLANK

Contents

1 Introduction

The ELLA^{TM} system is an integrated hardware design tool-set, which comprises the ELLA language compiler, the ELLA Applications Support Environment (EASE), the ELLA simulator, and the ELLANET procedural interface [Com90a][Com90b][Com90c][Com90d]. The language is used to describe hardware at all stages in the VLSI design cycle, from the earliest architectural concepts to the full implementation at gate level. The ELLA system was originally developed at the Royal Signals and Radar Establishment in Malvern, UK. and is now being enhanced in collaboration with Computer General Electronic Design (ED) of Chippenham. ELLA is the de-facto standard high-level VLSI design language in the UK, and is marketed by ED. In 1989 ELLA won a Queens Award for Technological Achievement.

ELLA describes a circuit as a hierarchy of interconnected nodes or networks. Networks can be described using an explicit netlist style or an implicit functional form. The language has no predefined signals and therefore a user must define the most appropriate form of abstract data type for the design under consideration. The behaviour of the network is defined in terms of leaf nodes or expressions. Many of the higher level syntactic constructs in the language are directly replaceable by a series of lower level ones. This facility means that high level abstract behavioural designs can be transformed down into lower level circuits ready for acceptance by low level back end tools. Thus users interested in synthesis and/or verification can maintain a top level description of a circuit whilst experimenting with different transformations in order to get the optimum low level design.

ELLA has been developing at RSRE for the past ten years. Throughout this development the original language [MPT85] has been built upon, with many of the new features transformable into the original language [WMW+89][MPW87]. New constructs have also been added to the set of original core constructs, for example ELLA V4 has the ability to model real arithmetic as well as bit string manipulation [HPC+90]. This is possible by the introduction of a new form of data type and a set of Built in Operators (BIOPs). The language is currently being enhanced under several collaborative projects, and many of the new language features are described in [HWM90b][HWM91]. This work will culminate in the release of ELLA V6. Work described in [HWM90b] comprises the introduction of named output signals, multiple declarations and a greatly enriched mechanism for joining up circuits. Memorandum [HWM90b] also describes enhancements to the timing model for allowing multi-rate circuits, a formal definition of which is given in [HWM90a]. The memorandum [HWM91] describes the rationalisation to the function type mechanism. The ability to 'link-in' non-ELLA code, such as microprocessor models, and provide powerful parameterisable functions (these take function instantiations/templates as parameters) increases the languages flexibility. These enhancements will be described in [DCT] along with most of the recent language enhancements.

In this report we give the formal definition of the static semantics of ELLA's Core constructs. We start in section 2 by describing the set of constructs which form the Core of the language and then indicate how, by the means of a set of software and syntactic transformations, any circuit can be mapped into a set of Core constructs. Although Core ELLA is a simpler language than the full language it still needs to address the problems of scopes and type checking. To this end a set of data structures have been defined which removes such obstacles. This is achieved by introducing an environment which provides every identifier with a unique number. Thus scopes and types are effectively flattened so that the environment can identify anything by its unique representation. The set of data structures is defined to be the **Kernel** of ELLA

and it is possible to map any description in the full language down onto structures in the **Kernel**. The definition of the **Kernel** along with the formal transformations to go from Core ELLA to the **Kernel** are given in section 3. Since the **Kernel** is a set of data structures all the information it holds cannot easily be expressed in terms of a concrete syntax. However it is possible to consider subsets of the **Kernel** for which a syntax is possible, see for example [BGL+91].

The work described in this report has been carried out for an IED (Information Engineering Directorate) project on "Formal Verification Support for ELLA" in collaboration with Manchester University Computer Science Department and Harlequin Ltd. of Cambridge. The aims of the **Kernel** are therefore that it should provide a simple system for mathematical manipulation and that it should interface simply with the verification work of the IED project, which is being carried out by Manchester University Computer Science Department.

2 Core ELLA

2.1 Introduction

In this section we give the definition of Core ELLA. Core ELLA is a subset of the complete language which is small enough for formal definition and manipulation, yet large enough to retain the character and flavour of the complete language. In essence Core ELLA is that part of the language which is arrived at from a description in the full language by applying a set of software transformations and/or a set of formal syntactic transformations. This is possible since ELLA's evolution has, to a large extent, occurred by defining the semantics of the more sophisticated syntactic constructs in terms of the semantics of more primitive constructs. The language has thus been able to retain a small number of primitive orthogonal constructs as its nucleus, with the complex constructs transforming onto this nucleus. Thus Core ELLA, though much simpler than the complete language, contains the essence of the complete language.

We start by giving a brief outline of the software transformational system and then in the following subsection subdivide the full language into different classes to facilitate the definition of Core ELLA.

2.2 Software Transformational System

The Software Transformational System is a collection of software procedures which take high level language constructs and transform them into lower level constructs. The transformed circuit description can then be viewed by printing the result in ELLA textual form. This Transformational System has allowed the incorporation into the language of a number of sophisticated syntactic constructs by simply relating their semantic interpretation to already existing constructs. The present set of transforms can be broadly categorised into the following, **macro**, **sequence**, **step**, **function-type**, **timescale** and **imports**.

The **macro** transformation replaces calls of macros by the appropriate function instance with parameters substituted. It also transforms out replicators and evaluates constant and integer expressions. Although it is possible to separate the macro and replicator transforms into two distinct transforms they have been kept together because replicators are a particular form of macro. The **sequence** transformation replaces all the sequence constructs by appropriate lets, makes, joins and unit delays. The **step** transformation replaces all the multiple lets, makes and complex join statements by occurrences of their simpler versions. The **function-type** transformation removes all occurrences of function type signals, for example a single bidirectional wire is replaced by two distinct wires. Use of **timescale** transforms the hierarchical retiming constructs (i.e. FASTER and SLOWER) into functions with delays and a sample-and-hold construct, whilst **imports** substitutes the bodies of imported functions into their appropriate calling function.

As a simple example of the transformational system consider the following Set/Reset Latch which is described using ELLA's sequential constructs

```
TYPE bool = NEW ( t | f ).

FN SR_LATCH = (bool: set reset) -> [2]bool:
( SEQ
    PVAR q ::= (f,t);                  # create and initialise state variable  #
    CASE (set, reset)
    OF (t,t)|(f,t): q := (f, t),       # state variable assigned new value     #
          (t, f)  : q := (t, f),
          (f, f)  :                    # state variable retains previous value #
    ESAC;
    OUTPUT q                           # output new value of state variable    #
).
```

When this is passed through the transformations the sequence constructs are replaced by equivalent parallel constructs to produce the following circuit

```
FN F1_DELAY = (( bool, bool )) -> ( bool, bool ):DELAY(( f, t ), 1 ).

FN SR_LATCH = ( bool: set reset ) -> [ 2 ]bool:
   ( MAKE F1_DELAY : s4q.
     LET q = CASE ( set, reset )
              OF ( t, t ) | ( f, t ): ( LET s6q = ( f, t ).
                                         OUTPUT s6q
                                       ),
                 ( t, f )            : ( LET s7q = ( t, f ).
                                         OUTPUT s7q
                                       ),
                 ( f, f )            : s4q
             ELSE s4q
             ESAC.
     JOIN q -> s4q.
     OUTPUT q
   ).
```

where names with an 's' followed by a number are generated by the transformation process. It should be noted that the code generated has been designed for simulation purposes rather than readability and conciseness.

The amount of code generated by the transformations depends on how concisely the higher level description has been written. For example the following FIFO stack expands to over 100 lines of code when passed through the transformations. The complete transformed design is given in appendix E.

```
TYPE bool = NEW ( t | f | x ),
     int  = NEW i/(0..100).

MAC FIFO {INT size} = (int: data_in,  bool: shift_in, bool: shift_out)
                                    -> (int, bool):
(SEQ
      PVAR fifo ::= [size](i/0, f);

      CASE shift_out
         OF  t : fifo := fifo[2..size] CONC (i/0, f)
      ESAC;

      VAR entered := f;
      CASE shift_in
      OF  t : [INT i = 1..size-1]
                  CASE (entered, fifo[i][2])
                    OF  (f,f) : ( fifo[i] := (data_in, t);
                                  entered := t
                                )
                  ESAC
      ESAC;

      OUTPUT (fifo[1][1], entered)
).

FN FIFO_9 = (int: data_in, bool: shift_in, bool: shift_out) -> (int, bool):
FIFO {9} (data_in, shift_in, shift_out).
```

It is possible to synthesise ELLA circuits down to gate level by using one of the currently available synthesis systems, e.g. GATEMAP [Pitt88]. Appendix F gives an example of a circuit which has passed through the GATEMAP system.

In the next section we split the full language into three classes in order to aid the definition of the language features for Core ELLA.

2.3 ELLA Hierarchical Definition

For the purposes of this report the full language will be subdivided into three classes. The first of these classes contains those constructs for which a formal semantic definition will not be required

2.3.1 Excluded Constructs

There are five language constructs which come into this class and they each have a different reason for being selected

- ARITH (being superseded by BIOPs)
- FNSET (being superseded by extensions to function types)

- Attributes (these provide an interface to external software)
- ALIEN (Linkage to non-ELLA code)
- Machine Dependent BIOPs (Because of machine dependency)

It is probable that all the operations which can be performed with ARITH statements will be achievable through the BIOP mechanism and therefore there is no loss of functionality by including ARITH in this class of constructs. It should be noted that BIOP's also provide a more rigorous definition of arithmetic operations.

The function type extension [HWM91] allows function type signals in the output part of a function specification, this means that function sets can be represented by functions with function type output signals and therefore function sets need not be considered as a separate case.

The remaining three constructs either describe linkage to non-ELLA language code or code which depends on the platform on which ELLA is installed.

2.3.2 Core ELLA Constructs

This second class contains those constructs which will form Core ELLA. It would have been possible to reduce the size of Core ELLA even further if functions had been transformed out and only basic enumerated types allowed. However this would have left only a single large network of nodes for any context and such a network, whilst suitable for a simulator, would have been very unhelpful for a user of the IED projects' verification environment. In particular if a user experimented with rules built up from the semantics there would be a problem in relating information back to the user, such as what part of the original ELLA circuit was being analysed.

All the basic language primitives have been incorporated including the latest additions to the ELLA language i.e. those from the numerics package [HPC+90], retiming [HWM90a] and BIOP's [Tai88a]. Rows have been included since they provide useful functionality and they complement STRING's. Constant, integer and macro declarations are not included since they are evaluated before a circuit is assembled, via the software transformation system. Only single makes, joins and lets are necessary since the multiple version of these constructs can be transformed into the simple versions.

2.3.2.1 Types The complete ELLA typing system has been included, i.e.

- Enumerated declaration (e.g. bool, int, char)
- Associated declaration (e.g. tag&bool)
- Unknown value (e.g. ?type)
- Collaterals (e.g. (bool,bool))
- Rows (e.g. [3]bool)
- STRINGS (of ELLA characters e.g. STRING[3]chars)
- Void (e.g. ())

Thus all forms of enumerated types including ELLA integers and ELLA characters are available in Core ELLA. All possible forms of structured types are also available as well as an ELLA unknown value corresponding to each type. The 'Void' type has also been included and represents a non-value carrying type. Such a type can be used, for example, when a function does not require either an input or an output signal, as in the case of a test harness.

2.3.2.2 Constants and Integers Within Core ELLA all constants and integers will be given as explicit basic values. Since constant and integer expressions are always defined statically in ELLA and the **macro** transformation can simplify them to basic values this is not a severe limitation. Hence the type of expressions available are

- Constant enumerated values (e.g. true, i/2, c'z)
- Constant type names (e.g. bool)
- Constant associated type values (e.g. tag&true)
- Integer values (e.g. 5)

2.3.2.3 Primitive Nodes All the primitive nodes of ELLA are included in Core ELLA. These are

- CASE with no 'ELSEOF' (e.g. CASE bool OF t:f ELSE x ESAC)
- REPLACE (e.g. replace an element of an array)
- DELAY (e.g. Ambiguity Delay primitive)
- IDELAY (e.g. Inertial Delay primitive)
- RAM (e.g. Multiple-element read/write memory: RAM([256]i/1)).
- SAMPLE (e.g. the SAMPLE-and-hold timing primitive used for retiming)
- BIOP (e.g. machine independent Built-In-Operators)

It is necessary to include all these primitive nodes of ELLA since they are not easily transformed. The CASE statement has been restricted to exclude the ELSEOF alternative since the ELSEOF part can be transformed out (see section 2.3.3.2). The new REPLACE construct allows arrays to have one of its fields replaced by the output from a value delivering expression. The two forms of Delays are needed since they are fundamental in providing ELLA's functionality. The new sample-and-hold primitive has been included so that designs with retiming can be transformed into Core ELLA. Only those BIOP's which are machine independent are included here.

2.3.2.4 Functions Function declarations have been retained within Core ELLA in order to preserve the modularity of a design. It would be possible to transform out all functions within a context but this would leave only a flattened circuit. Most designers use functions in order to partition designs into more manageable units and this feature is felt to be desirable for Core ELLA. By allowing functions it also means that operations on them can be included e.g. make, join. The complete list of function related constructs is

- Functions Declarations

- MAKE (e.g. MAKE AND: and.)
- Implicit Monadic Function Calls (e.g. AND(in1,in2))
- JOIN (e.g. JOIN ... → name)
- LET (e.g. LET name = ...)
- BEGIN...OUTPUT...END clause
- Locally declared function and type declarations

Only monadic function calls have been included since a dyadic call is just another way of representing a monadic function with two inputs. Both the JOIN and LET statements are restricted to the ELLA V4 syntax [Com90a] since the enhancements recently carried out [HWM90b] can be transformed into that form. The BEGIN...END clauses is included in order to allow the LET, JOIN, MAKE statements to be used, as well as to declare local functions and types.

2.3.2.5 Signal Structuring and Extraction Signals within a Core ELLA program will need to be structured, or extracted, from other signals. For this reason the following have been included in Core ELLA.

- Indexing (e.g. id[2])
- Trimming (e.g. id[1..6])
- Dynamic Indexing (e.g. indexing an expression by a signal value)
- CONC (concatenation)
- REFORM (of signal groups)
- Associated type constructs (i.e. '//' and '&')
- Rows/STRINGS of value delivering clauses
- Collaterals of value delivering clauses

Indexing, dynamic indexing and trimming enables components of structures to be extracted and CONC enables components to be combined. REFORM is a means by which a set of signals can be regrouped to form a different combination e.g. (bool,[3]bool) can be reformed to ([2]bool,[2]bool). The two associated type constructors are needed since there is no other way for creating and extracting data from an associated type. The last two items are ways in which structures of value delivering clauses can be constructed.

2.3.3 Transformable Constructs

The last class contains the remainder of the ELLA language. It should be noted that some of the features in this list are not available in the commercial release ELLA V4 [Com90a] but they are available in an RSRE version [HWM90b] [HWM91] which will form ELLA V6, see appendix B. Most of these language features are already transformed by software into a series of constructs in Core ELLA (see section 2.2). Thus although at first sight this class of language features might appear to encompass a large amount of ELLA, all language features up to and including those of Release 6 will be transformable into Core ELLA.

2.3.3.1 Software Transformations The complete list of constructs for which software transformations exist is

- Redundant brackets (e.g. ((bool)))
- IF boolean THEN unit ELSE unit FI
- Named outputs (e.g. FN A = (bool) → (bool:out):...)
- Multiple lets (e.g. LET (a1, a2) = ...)
- Multiple makes (e.g. MAKE [2][3][2]A: a.)
- Complex joins (e.g. JOIN (true,false) → (a1, a2[2]).)
- Replicated joins (e.g. FOR INT i = 1..2 JOIN ...)
- Replicators ([INT i = 1 .. n])
- INT declarations (e.g. INT i = 6)
- CONST declarations (e.g. CONST c = (true, (true |false)).)
- Sequences (e.g. VAR, PVAR etc.)
- Macros (with INT, TYPE, CONST, FN, MAC parameters)
- Function types (bidirectional wires e.g. fntype = bool → [2]bool)
- IO (supplying the Input and Output of a function type: used in 'joining')
- FASTER/SLOWER (hierarchical retiming constructs e.g. speed up/slow down)
- Print/Fault (macro expansion assertions)
- Imports/exports (control of functions through multiple contexts)
- Renamed (renaming of imported function)
- Static operators (e.g. int/(1+(m*20))
- Naming of previously unnamed input terminals
- Abbreviated BEGIN..END clauses (e.g. (... OUTPUT ..))
- Series of declarations separated by commas converted into separate declarations
- Transport Delays converted into equivalent ambiguity delays

A couple of constructs which cannot be transformed by means of the software transformations have been excluded from the definition of Core ELLA. These constructs can however be supplied with ELLA to ELLA syntactic transformations which define their behaviour in terms of equivalent constructs which can be transformed into Core ELLA.

2.3.3.2 Syntactic Transformations As mentioned in the previous section there are constructs which have been excluded from Core ELLA for which software transformations from the complete language do not exist. These constructs are

- CASE statements with ELSEOF alternatives
- Integer and character ranges in CASE choosers
- Character ranges in TYPE declarations

All of these constructs can be transformed into versions which are acceptable to the Core. For example the first item can have the ELSEOF parts removed, and the other two items can have the ranges expanded out. The definition of syntactic transformations for these constructs can be given in terms of re-write rules

$$\boxed{\text{ELLA}}\ \frac{\text{language feature}}{\text{semantically equivalent language features}}$$

where the 'language feature' above the line is semantically equivalent to the set of language features below the line.

The following syntax transformation rules for the above constructs are defined on the full language, the complete syntax for the full language being given in appendix B.

$$\boxed{\text{ELLA-Case1}}\ \frac{\begin{array}{c}\text{elsof}_i = \text{ELSEOF cases}_i \qquad \text{ForAll } i \in \{1..k\} \\ \text{CASE unit OF cases } \mathit{elseof}_1 \cdots \mathit{elseof}_k \text{ ESAC}\end{array}}{\begin{array}{c}\text{CASE unit OF cases elseof}_1 \cdots \text{elseof}_{k\text{-}1} \\ \text{ELSE CASE unit OF cases}_k \text{ ESAC ESAC}\end{array}}$$

$$\boxed{\text{ELLA-Case2}}\ \frac{\begin{array}{c}\text{elsof}_i = \text{ELSEOF cases}_i \qquad \text{ForAll } i \in \{1..k\} \\ \text{CASE unit}_1 \text{ OF cases elseof}_1 \cdots \text{elseof}_k \text{ ELSE unit}_2 \text{ ESAC}\end{array}}{\begin{array}{c}\text{CASE unit}_1 \text{ OF cases elseof}_1 \cdots \text{elseof}_{k\text{-}1} \text{ ELSE} \\ \text{CASE unit}_1 \text{ OF cases}_k \text{ ELSE unit}_2 \text{ ESAC ESAC}\end{array}}$$

$$\boxed{\text{ELLA-Range1}}\ \frac{\text{tagname / (lwb .. upb)}}{\text{tagname / lwb } | \cdots \ | \text{tagname / lwb + (upb-lwb)}}$$

$$\boxed{\text{ELLA-Range2}}\ \frac{\text{tagname ('firstchar .. 'lastchar)}}{\text{tagname 'firstchar } | \cdots | \text{ tagname 'lastchar}}$$

$$\boxed{\text{ELLA-Range3}}\ \frac{\text{NEW tagname (} \cdots \ | \text{ 'firstchar .. 'lastchar } | \cdots \text{)}}{\text{NEW tagname (} \cdots \ | \text{ 'firstchar } | \cdots | \text{ 'lastchar } | \cdots \text{)}}$$

Having defined this set of syntactic transformations we can now proceed to define the syntax for Core ELLA.

2.4 Core ELLA Composite Syntax

As defined in [BHM90] a composite syntax combines the essences of both the concrete and abstract syntax of a language. The composite syntax has been defined to retain the binding of the concrete syntax, thus it accurately represents the binding of the ELLA parser. The composite syntax defined in this section is summarised in appendix C .

In order to define the composite syntax a set of syntactic categories are need. Their definition has been chosen in order to maximise the readability of the syntax. Before proceeding to define the categories some basic notation will be given

2.4.1 Basic Notation

Throughout this section the following notation will be used

abc $\in$ Abc	$\equiv$	'abc' is an element of the set 'Abc'
b ::= c	$\equiv$	the syntax definition of 'b' is 'c'
$\vert$	$\equiv$	the separator of alternatives in a syntax definition
$\boldsymbol{\vert}$	$\equiv$	ELLA separator of alternatives
$d_1 \cdots d_k$	$\equiv$	one or more occurrences of 'd'
$d_1, \cdots, d_k$	$\equiv$	one or more occurrences of 'd' separated by ','. Note if k=1 then no ',' is present.
$d_1, \cdots, d_{k-1}$	$\equiv$	zero or more occurrences of 'd' separated by ','. Note if k=0 then no ',' is present.

2.4.2 Syntactic Categories

The categories used throughout this section are described below. Words which begin with an upper case letter represent Sets, where 'Identifier' is the set of all lower case names, 'Fnname' the set of all upper case names and symbols. Note that 'Identifier' and 'Fnname' are disjoint Sets. 'Character' is the Set of all printable characters, whilst 'String' is the set of all character strings composed from elements of 'Character'. The Sets '$\mathbb{Z}$' and '$\mathbb{N}_1$' are the Sets of integers and natural non-zero numbers respectively.

typename	$\in$	Identifier	(ELLA type name e.g. lower case)
signalname	$\in$	Identifier	(ELLA signal name)
tagname	$\in$	Identifier	(ELLA tagged type name)
altname	$\in$	Identifier	(ELLA enumerated type alternative)
fnname	$\in$	Fnname	(ELLA function name e.g. upper case or symbol)
biopname	$\in$	Fnname	(ELLA BIOP name e.g. upper case or symbol)
z	$\in$	$\mathbb{Z}$	(An integer)
lwb, upb	$\in$	$\mathbb{Z}$	(An integer)
j,k	$\in$	$\mathbb{N}_1$	(A non-zero positive integer)
index	$\in$	$\mathbb{N}_1$	(A non-zero positive integer)
size	$\in$	$\mathbb{N}_1$	(A non-zero positive integer)
interval	$\in$	$\mathbb{N}_1$	(ELLA timing interval)
ambigtime	$\in$	$\mathbb{N}_1$	(Ambiguity delay time)
delaytime	$\in$	$\mathbb{N}_1$	(delay time)
skewtime	$\in$	$\mathbb{N}_1$	(skew delay)

con	$\in$	Constant	(**'con' is a value of the ELLA Constant type 'Constant'**)
initialvalue	$\in$	Constant	(**Delay, Retiming or Ram initialisation value**)
ambigvalue	$\in$	Constant	(**Delay ambiguity value**)
char	$\in$	Character	(**A printable character e.g. 'a'**)
string	$\in$	String	(**A string of printable characters e.g. 'abc'**)

Some of these names have been taken from the abstract syntax (i.e. delay parameter names) in order to aid readability. We now give the definition of the Core syntax.

2.4.3 Enumerated Values

The following four basic enumerated values will be grouped together

```
enumerated   ::=    altname
                  | tagname / z
                  | tagname 'char
                  | tagname "string"
```

The symbols /, ', " are ELLA defined symbol tags which enable the 'tagname' to be correctly associated with the appropriate ELLA type. It can be noted that the basic associated type value is missing from the list since the associated part is either a constant, unit or type.

2.4.4 Types

A type in Core ELLA is defined to be

```
type         ::=    typename
                  | STRING [ size ] typename
                  | [ size ] type
                  | ( type1, ···, typek )
                  | ()
```

where 'typename' is the name of a type declaration (see below). Note that the collateral of types can have just one element which therefore allows redundant brackets in types. The constant 'size' is taken to have a known positive integer value.

2.4.5 Constants

Core ELLA constants have been split into two classes in order to preserve the binding of the concrete syntax.

```
const        ::=    STRING [ size ] const1
                  | [ size ] const
                  | const1

const1       ::=    enumerated
                  | altname & const1
```

	|	($\text{const}_1, \cdots, \text{const}_k$)
	|	? type
	|	()

Constants defined by this syntax will be used as initialising values in delays and rams.

2.4.6 Constant Sets

Constant Sets are collections of constants which are used as choosers in CASE statements. The definition of Constant Sets differs from that of Constants in two ways. First the inclusion of the top level definition which allows for constant alternatives. Second the last option of 'constset2' is 'type' and not '?type'. The last change is necessary since it is not possible to test for '?type' in a CASE statement. The full definition of Constant Sets is

constset	::=	$\text{constset}_1 \mid \cdots \mid \text{constset}_k$
constset1	::=	STRING [size] constset2
	|	[size] constset1
	|	constset2
constset2	::=	enumerated
	|	altname & constset2
	|	($\text{constset}_1, \cdots, \text{constset}_k$)
	|	type

2.4.7 *Units*

The syntax element 'unit' is the basic building block for all Core ELLA descriptions. Units supply single or multiple values and are composed of either basic values or clauses which deliver basic values. The definition given here has three levels which represent the levels of binding given in the concrete syntax.

unit	::=	unit CONC unit1
	|	unit1
unit1	::=	STRING [size] unit1
	|	[size] unit1
	|	fnname unit1
	|	altname & unit1
	|	unit2 // altname
	|	unit2
unit2	::=	signalname
	|	enumerated
	|	unit2 [index]
	|	unit2 [index_{lwb} .. index_{upb}]
	|	unit2 [[unit]]
	|	REPLACE (unit, unit, unit)
	|	? type
	|	closedclause

The last alternative of 'unit2' is a closedclause which is defined in the next section. The 'signalname' of 'unit2' is either a MAKE, LET or parameter name. The constants 'index' are known integer values.

2.4.8 Closedclause

Closedclauses provide the mechanism for combining units. The CASE statement provides a multiplexer-type construct, and the BEGIN...END clause gives a method for signal naming and introducing local declarations.

closedclause	::=	CASE unit OF cases ELSE unit ESAC
	|	($\text{unit}_1, \cdots, \text{unit}_k$)
	|	BEGIN $\text{step}_1 \cdots \text{step}_{k\text{-}1}$ OUTPUT unit END
	|	()

cases	::=	$\text{constset}_1 : \text{unit}_1, \cdots, \text{constset}_k : \text{unit}_k$

step	::=	typedec
	|	fndec
	|	LET signalname = unit .
	|	MAKE fnname : signalname .
	|	JOIN unit → signalname .

Note that within a BEGIN ··· END clause there need not be any 'step's. The use of $\text{step}_{k\text{-}1}$ implies that if k = 1 then no 'step's are present. It can be noted that the 'signalname's are handled differently in the different 'step' declarations. For example, the 'signalname' in a LET names the output of the 'unit' and hence it can only be used to supply a signal to some other part of a circuit. This means that such a signalname could not occur on the right hand side of the arrow in a JOIN statement. Whereas the signalname in the makedecs is naming a function instantiation and hence has both a value-requiring part and a value-delivering part. This means such a signalname could be used on either side of the '→' in a JOIN statement.

2.4.9 Function Body

A function body can either be described by a 'unit' or by a function body primitive.

functionbody	::=	unit
	|	REFORM
	|	BIOP biopname
	|	DELAY (initialvalue, ambigtime, ambigvalue, delaytime)
	|	IDELAY (initialvalue, delaytime)
	|	SAMPLE (interval, initialvalue, skewtime)
	|	RAM (initialvalue)

All the basic syntactic definitions of Core ELLA have now been completed. The three remaining classes define how the above information is combined to form a Core ELLA closure.

2.4.10 Type Declarations

A Type declaration is of the following form

typedec	::=	TYPE typename = typeornew.
typeornew	::=	type
	|	new
new	::=	NEW tagname / (lwb .. upb)
	|	NEW (typealt_1 | ⋯ | typealt_k)
	|	NEW tagname ($'\text{char}_1$ |⋯ |$'\text{char}_k$)
typealt	::=	altname & type
	|	altname

where the alternatives of 'typeornew' represent: the name of a previous type declaration, or an alternative of 'new' i.e. an ELLA integer, an enumerated type, or an ELLA character. The enumerated type allows for some of the alternatives to have associated values.

2.4.11 Function Declarations

A function declaration is given by

fndec	::=	FN *fnname* = input → type : functionbody.
input	::=	(type_1 : signalname_1, ⋯, type_k : signalname_k)
	|	()

2.4.12 Closure

A Core ELLA Closure is the entry point into a Core ELLA context and consists of a number of type and function declarations.

declaration	::=	typedec
	|	fndec
closure	::=	$\text{declaration}_1 \cdots \text{declaration}_k$

In any closure declarations are built upon previously defined declarations. Thus declaration_i can only use declaration_l where $1 \leq l < i$. Hence declaration_k is considered to be the top level declaration.

2.5 Well Formedness of Core ELLA

In this section we consider the well formedness of a description in Core ELLA. A well formed description means that the transformation rules from Core ELLA to the **Kernel** can be successfully applied. The transformation rules described in section 3.5 effectively give the conditions that a Core description must satisfy in order for the transformations to produce a result. What is missing from the transformation rules is any indication of what happens when conditions are not fulfilled, and this raises the question of error detection. Throughout the development of ELLA great emphasis has been placed on providing an error detection and reporting system that gives the designer the maximum amount of information possible in order to facilitate correction of errors. This results in a significant amount of the compiler's time being taken up with the monitoring and handling of information necessary for the error reporting system. Inevitably if a similar error reporting structure is implemented for the transformations to the **Kernel** the number of rules would increase significantly. Thus, for the present, we shall merely consider the different categories of errors that such a transformational system would need to address.

The different categories of errors needed for the transformations to the **Kernel** can best be seen by observing an abstract syntax for Core ELLA. Consider the following data structure (see [SWH88] for a complete description of an abstract syntax for the full language)

Closure ::= Declaration_1 ... Declaration_k

Declaration ::= $\textbf{typedec}_1(\text{Id} \times \text{Typebody})$
| $\textbf{fndec}(\text{Id} \times \textbf{terminals}(\text{Id}_1 \times \text{Type}_1 \times \ldots \times \text{Id}_m \times \text{Type}_m) \times \text{Type}_o \times \text{Fnbody})$

Typebody ::= Type
| $\textbf{alternative}_1(\text{Id} \times \text{Type}) \ldots \textbf{alternative}_k(\text{Id} \times \text{Type})$
| $\textbf{newchars}(\text{Id} \times \textbf{chars}(\text{Char}_1 \times \ldots \times \text{Char}_k))$
| $\textbf{newints}(\text{Id} \times \text{Int} \times \text{Int})$
| **tbody-fail**

Type ::= $\textbf{tname}(\text{Int})$
| $\textbf{tstr}(\text{Type}_1 \times \ldots \times \text{Type}_k)$
| $\textbf{trow}(\text{Int} \times \text{Type})$
| $\textbf{tstring}(\text{Int} \times \text{Type})$
| **tnull**
| **tvoid**
| **type-fail**

Const ::= **cbasic**(Int × Int)
| **crow**(Int × Const)
| **cstring**(Int × Const)
| **cassoc**(Int × Int × Const)
| **cstr**(Const_1 × ... × Const_k)
| **cquery**(Type)
| **cchar**(Int × Char)
| **cquote**(Int × Char_1 × ... × Char_k)
| **cvoid**
| **const-fail**

Constset ::= **csalts**(Constset_1 × ... × Constset_k)
| **csbasic**(Int × Int)
| **csrow**(Int × Constset)
| **csstring**(Int × Constset)
| **csassoc**(Int × Int × Constset)
| **csstr**(Constset_1 × ... × Constset_k)
| **cstype**(Type)
| **cschar**(Int × Char)
| **csquote**(Int × Char_1 × ... × Char_k)
| **constset-fail**

Unit ::= **cbasic**(Int × Int)
| **cquery**(Type)
| **cchar**(Int × Char)
| **cquote**(Int × Char_1 × ... × Char_k)
| **uname**(Int)
| **uassoc**(Int × Int × Unit)
| **uextract**(Unit × Int × Int)
| **uindex**(Unit × Int)
| **utrim**(Unit × Int × Int)
| **udyindex**(Unit × Unit)
| **ureplace**(Unit × Unit × Unit)
| **ustr**(Unit_1 × ... × Unit_k)
| **urow**(Int × Unit)
| **ustring**(Int × Unit)
| **uconc**(Unit × Unit)
| **uminst**(**instance**(Id) × Unit)
| **ucase**(Unit × **uchoices**(Const_1 × Unit_1 × ... × Const_k × Unit_k))
| **useries**(**series**(Step_1 × ... × Step_k) × Unit)
| **uvoid**
| **unit-fail**

Step	::=	**typedec**(Id × Typebody)
	|	**fndec**(Id × **terminals**(Id_1 × $Type_1$ × ... × Id_m × $Type_m$) × $Type_o$ × Fnbody)
	|	**let**(Id × unit)
	|	**make**(**instance**(Id) × Id)
	|	**join**(Unit × Id)
	|	**step-fail**
Fnbody	::=	Unit
	|	**reform**
	|	**adelay**(Const × Int × Const × Int)
	|	**idelay**(Const × Int)
	|	**ram**(Const)
	|	**sample**(Int × Const × Int)
	|	**biop**(Id)
	|	**fnbody-fail**

Where the constructors **tbody-fail**, **type-fail**, **const-fail**, **constset-fail**, **unit-fail**, **step-fail** and **fnbody-fail** represent the different classes of failure. The closure does not have an explicit failure constructor since any failure will occur in either a **typedec** or a **fndec** and the appropriate field of those constructors would be set to failure.

Whilst it would be relatively straightforward to obtain a set of transformation rules which were the negation of the rules in section 3.5, the interaction of the rules in the case of errors would require significant extra analysis.

3 Kernel ELLA

3.1 Introduction

In this section we present the **Kernel** of ELLA. The **Kernel** is a set of data structures which can describe the complete language. The **Kernel** is not itself a language since it does not have the problems of scopes or type checking; these are all taken care of by the transformations from Core ELLA to the **Kernel**. Within the **Kernel** other simplifications over the Core have also been made. For example all rows are converted to structures, and local BEGIN..END clauses are removed by means of the use of a transformation environment. This environment collects together all the type and function declarations and provides a means of referencing declarations via a global numbering system. As a consequence the local type and function declarations can also be removed to the outer level. Within each function declaration the environment also collects together all signal declarations and gives them a unique reference number. Another simplification that the environment allows is that the MAKE/JOIN information is added to the environment in a similar way to an implicit function call and hence this avoids the need for explicit MAKE/JOINs appearing within the **Kernel**.

Within the complete language there is a system of scope rules for allowing declarations to use identical names in separate parts of a closure. The transformational environment removes all scopes by ensuring that each type and function name used in a closure, and each signal name used in a function, has a unique reference number. This number then points into a declaration field of the environment. The rules governing this for a BEGIN..END clause work in the following way when transformation rule [CC3] is applied (see section 3.4.4 for scoping rule functions, and section 3.7.3 for an example of their use): When a BEGIN is encountered, each local map in the new environment is set to null, and each non-local map is set to the map constructed by combining the local and non-local maps of the previous environment. In the case of common arguments, the local entry overrides the non-local one. Then throughout the ensuing BEGIN..END clause the local map gets made up with items that are local to the BEGIN..END clause. At the END the local function names and type names will go out of scope and the declarations that were local prior to entry to the BEGIN..END come back into local scope. Thus on leaving the BEGIN..END the new environment takes the first three fields from the local environment and all the other fields from the environment prior to the BEGIN..END, except for those local names which are now out of scope and must therefore be added to the "used" fields to ensure that they are not redefined.

The **Kernel** has been designed in its present format so that it should provide a system for mathematical manipulation and that it should interface simply with the verification work of the IED project on "Formal Verification Support for ELLA". Although all the scopes have been removed, function hierarchy has not, even though function declarations are now all global. This means that a design can retain its structure. In the next section we describe the **Kernel** data structures, with the transformations from Core ELLA to the **Kernel** being dealt with in subsequent sections. The complete **Kernel** data structures are summarised in appendix D

3.2 Conventions

Throughout this section the following conventions will be observed:-

abc	$\in$	Abc (ie. it is an element of the set Abc)
Indexer, Size, Fnno	$\subseteq$	$\mathbb{N}_1$
Typeno, Tagno, Inputno	$\subseteq$	$\mathbb{N}_1$
Signalno, Ambigtime, Delaytime	$\subseteq$	$\mathbb{N}_1$
Interval, Skew	$\subseteq$	$\mathbb{N}_1$
Inputtype, Outputtype	$\subseteq$	Type
Initialvalue, Ambigvalue	$\subseteq$	Const
Inputfnspec	$\subseteq$	"Input of function specification"
Fnname, Biopname	$\subseteq$	"Upper case identifier or operator"
Name, Signalname	$\subseteq$	"Lower case identifier"
Typename, Tagname	$\subseteq$	"Lower case identifier"
Lowerbound, Upperbound	$\subseteq$	"positive or negative integer"
Character	$\subseteq$	"printable character"
kId	$\underline{\Delta}$	"object "Id" belongs in the **Kernel**"
a.b	$\underline{\Delta}$	"Field selector, (i.e. field "b" of "a")"
a[int]	$\underline{\Delta}$	"Indexing, (i.e. element "int" of "a")"

with appendix A collecting together the basic mathematical notation used throughout this section.

In the definition of the **Kernel** data structure, the following naming conventions are also used:

$abc_1, abc_2, abc_3, \cdots$	$=$	abc	separate instances of tuple *abc*
$AbcSeq$	$=$	$Abc \times \cdots \times Abc$	non-empty sequence of elements of type *Abc*
$AbcOpt$	$=$	$Abc \mid \{nil\}$	type Abc with optional element nil

One of the reasons for adopting these conventions is that names of component types are used as field selectors for structured types. This means that every field needs to be indicated by a unique name.

3.2.1 Links to VDM

It can be noted that the data structures definitions below naturally correspond to type definitions in the style of VDM. This correspondence is illustrated by means of the following extract of the **Kernel**

$$
\begin{array}{lll}
\text{Enumerated} & ::= & \text{Enum} \\
 & \mid & \textbf{string}(\text{ Typeno} \times \text{TagnoSeq }) \\
 & & \\
\text{Enum} & ::= & \textbf{enum}(\text{ Typeno} \times \text{Tagno }) \\
 & & \\
\text{Closure} & ::= & \text{TypedecSeq} \times \text{FndecSeq}
\end{array}
$$

which corresponds to the following in VDM:

$$Enumerated = Enum \mid String$$

$$
\begin{array}{lrcl}
String :: & typeno & : & Typeno \\
 & tagnoseq & : & Tagno^{+}
\end{array}
$$

Enum :: *typeno* : *Typeno*
tagno : *Tagno*

Closure = $Typedec^{+} \times Fndec^{+}$

The formal transformation rules and associated functions given in subsequent sections use VDM notation [Jon90] to define their functionality.

3.3 Kernel Data Structure

In this section we describe the data structure model of the **Kernel**.

Note that all names in the **Kernel** are referenced by integers. These are unique even though the scopes have been removed and their identifiers are no longer unique. These integers index the first three fields of the environment and the names are merely attributes of the indexed objects. For example 'Signalno' in **signal** indexes the *sigdec* field of the environment and the original name is obtained from the Signalname field of this object.

3.3.1 Enumerated Values

Enumerated values in the **Kernel** are defined as

Enumerated ::= Enum
| **string**(Typeno × TagnoSeq)

Enum ::= **enum**(Typeno × Tagno)

where Enum represents all the different enumerated types except for STRING types. Note that for integer ranges Tagno=1 corresponds to the lower bound of the integer range, even in those cases where the lower bound is negative.

3.3.2 Types

Types in the **Kernel** are defined by the following structures

Type ::= **typeno**(Typeno)
| **typename**(Typename × Type)
| **stringtype**(Size × Type)
| **types**(TypeSeq)
| **typevoid**

3.3.3 Constants

Constants are the representation of the constant class of Core ELLA which are used for initialisation of delays etc. and are defined by

Const	::=	Enumerated
	|	**conststring**(Size × Const)
	|	**consts**(ConstSeq)
	|	**constassoc**(Enum × Const)
	|	**constquery**(Type)
	|	**constvoid**

3.3.4 Constant Sets

Constant sets are similar to constants except that they allow sets of alternatives (i.e. **true** | **false**) and are used as choosers in case statements

Constset	::=	Enumerated
	|	**constsetalts**(ConstsetSeq)
	|	**constsetstring**(Size × Constset)
	|	**constsets**(ConstsetSeq)
	|	**constsetassoc**(Enum × Constset)
	|	**constsetany**(Type)

3.3.5 Units

The basic building blocks of a **Kernel** description are the Unit expressions which are defined by

Unit	::=	Enumerated
	|	**conc**(Unit × Unit × Outputtype)
	|	**unitstring**(Size × Unit)
	|	**units**(UnitSeq)
	|	**instance**(Fnno × Unit)
	|	**unitassoc**(Enum × Unit)
	|	**extract**(Unit × Enum)
	|	**signal**(Signalno)
	|	**index**(Unit × Indexer × Outputtype)
	|	**trim**(Unit × Indexer × Indexer × Outputtype)
	|	**dyindex**(Unit × Unit × Outputtype)
	|	**replace**(Unit × Unit × Unit)
	|	**unitquery**(Type)
	|	**caseclause**(Unit × CaseSeq × Unit)
	|	**unitvoid**

where the different alternatives in Case clauses are given by

Case	::=	**case**(Constset × Unit)

3.3.6 Function Declarations

A function declaration has the following data structure

Fndec	::=	**fndec**(Fnname × Inputtype × SignaldecSeq × Outputtype × Fnbody)

where

Signaldec	::=	**signaldec**(Signalname × Type × Unitorinput)
Unitorinput	::=	Unit
	|	**input**

and the Function Body is given by

Fnbody	::=	Unit
	|	**reform**
	|	**biop**(Biopname)
	|	**delay**(Initialvalue × Ambigtime × Ambigvalue × Delaytime)
	|	**idelay**(Initialvalue × Delaytime)
	|	**sample**(Interval × Initialvalue × Skew)
	|	**ram**(Initialvalue)

3.3.7 Type Declarations

A type declarations are of the form

Typedec	::=	**typedec**(Typename × New)

where

New	::=	**tags**(TagSeq)
	|	**ellaint**(Tagname × Lowerbound × Upperbound)
	|	**chars**(Tagname × CharacterSeq)
Tag	::=	**tag**(Tagname × TypeOpt)

3.3.8 Closure

Finally a closure is represented by

Closure	::=	TypedecSeq × FndecSeq

3.4 Environments and Signatures

This section describes the transformational environment and the signatures on Core ELLA, these are used for defining the transformations from Core ELLA to the Kernel.

3.4.1 Transformation Environment

The environment (*Env*) is defined to be a record object with 12 fields which will accumulate type, function and signal declarations and maintain information about the scopes of identifiers.

$$
\begin{array}{lrcl}
Env :: & typedec & : & kTypedec^* \\
 & fndec & : & kFndec^* \\
 & sigdec & : & kSignaldec^* \\
 & fnmap & : & Fnname \xrightarrow{m} Fnno \\
 & lclfnmap & : & Fnname \xrightarrow{m} Fnno \\
 & tynamemap & : & Name \xrightarrow{m} Typetag \\
 & lcltynamemap & : & Name \xrightarrow{m} Typetag \\
 & signamemap & : & Signalname \xrightarrow{m} Sig \\
 & lclsignamemap & : & Signalname \xrightarrow{m} Sig \\
 & usedtyname & : & Name\text{-}set \\
 & usedfnname & : & Fnname\text{-}set \\
 & usedsigname & : & Signalname\text{-}set
\end{array}
$$

$$
\begin{aligned}
& inv(Env) \ \triangleq \\
& \quad (\mathsf{dom}\,(Env.lcltynamemap) \cap \mathsf{dom}\,(Env.lclsignamemap) \cap \mathsf{dom}\,(Env.signamemap) \\
& \qquad \cap\, Env.usedsigname \cap Env.usedtyname = \{\,\}) \\
& \qquad\quad \wedge (\mathsf{dom}\,(Env.lclfnmap) \cap Env.usedfnname = \{\,\})
\end{aligned}
$$

with the following being local to the translation process

Typetag	=	**typeno**(*Typeno*)	(new TYPE)
		∪ **typename**(*Typename* × *kType*)	(TYPE alias)
		∪ **consttag**(*Typeno*)	(TYPE tagname alternative)
Sig	=	**sig**(*Signalno* × *Sort*)	(Signal name)
Sort	=	**joined** | **unjoined**	(status of signal input field)

The invariant of the environment is defined to be $inv(Env)$ which states that all signal and type names must be unique and all function names must be unique.

Note that the first three fields of *Env* are sequences. The use of each field can be summarised as follows

Env.typedec	Accumulates all typedecs for the final closure
Env.fndec	Accumulates all fndecs for the final closure
Env.sigdec	Accumulates signaldecs for each fndec
Env.fnmap	Fn name map - visible outside the most local scope,
Env.lclfnmap	Fn name map - in most local BEGIN..END scope,
Env.tynamemap	Type information map - visible outside the most local scope,
Env.lcltynamemap	Type information map - in most local BEGIN..END scope,
Env.signamemap	Signal name map - visible outside the most local scope,
Env.lclsignamemap	Signal name map - in the most local BEGIN..END scope,
Env.usedtyname	Type names out of scope but unavailable for reuse
Env.usedfnname	Fn names out of scope but unavailable for reuse
Env.usedsigname	Signal Names out of scope but unavailable for reuse

Note *fnname*'s are generated by FN declarations, *tyname*'s are generated by TYPE declarations (both the TYPE name and their tags), and *signame*'s are generated by MAKE, LET and input parameter declarations. The fields '*used...name*' are the set of all names which have been used in the current FN but are now out of scope. This field is used by the '*Check*' functions to disallow their redefinition. This is a deliberate restriction in the full language to ensure unique path names in function bodies.

The initial environment contains only empty declarations i.e.

$$InitialEnv = \mathbf{env}([\,],[\,],[\,],\{\,\},\{\,\},\{\,\},\{\,\},\{\,\},\{\,\},\{\,\},\{\,\},\{\,\})$$

3.4.2 Built-In Operator Environment

The environment for the Built-In Operators (*BiopEnv*) is a sequence of objects which hold the BIOP name and its typing information

$$BiopEnv :: biop : kBiop^*$$

$$\begin{aligned} Biop :: \; biopname &: Fnname \\ inputtype &: kType \\ outputtype &: kType \end{aligned}$$

The *BiopEnv* is chosen at the outset and remains fixed for the complete transformation process.

3.4.3 Signatures

Throughout this section a number of signatures will be needed. The complete list is given by

Scope-Fn-Begin	:	Env ⟶ Env
Scope-Fn-End	:	Env × Env ⟶ Env
Scope-Begin	:	Env ⟶ Env
Scope-End	:	Env × Env ⟶ Env
Check-Joins	:	Env ⟶ $\mathbb{B}$
Check-Two-Val	:	kType ⟶ $\mathbb{B}$
Check-Fn	:	Fnname × Env ⟶ $\mathbb{B}$
Check-Typename	:	Name × Env ⟶ $\mathbb{B}$
Check-Signal	:	Signalname × Env ⟶ $\mathbb{B}$
Add-Fn	:	Fndec × Env × Env ⟶ Env
Add-Type	:	Typedec × Env ⟶ Env
Add-Signal	:	Signaldec × Sort × Env ⟶ Env
Add-Join	:	Signaldec × Signalno × Env ⟶ Env
Add-Tag	:	Tagname × Env ⟶ Env
Add-Type-Name	:	Typename × kType × Env ⟶ Env
Find-Type-Nm	:	Name × Env ⟶ Typetag
Find-Sig-Nm	:	Name × Env ⟶ Sig
Find-Fn	:	Fnname × Env ⟶ Fnno
Find-Unjoined	:	Signalname × Env ⟶ Fnno
Find-Type	:	Typename × Env ⟶ kType
Find-Alt	:	Altname × Env ⟶ kEnum
Find-ELLAint	:	Tagname × Env ⟶ Typeno × Lowerbound × Upperbound
Find-Integer-Type	:	kType × Env ⟶ Typeno × Lowerbound × Upperbound
Find-Char	:	Tagname × Char × Env ⟶ kEnum
Find-Signal	:	Signal × Env ⟶ kUnit × kType
Find-Assoc	:	Altname × Env ⟶ kType
Find-Row	:	kType ⟶ $\mathbb{N}_1$ × kType
Find-Biop	:	Biopname × kType × kType ⟶ kFnbody

Get-Type	:	**kType** $\longrightarrow$ **kType**
Type-Equals	:	**kType** $\times$ **kType** $\longrightarrow \mathbb{B}$
Conc	:	**kType** $\times$ **kType** $\longrightarrow$ **kType**
Is-Char	:	$\mathbb{N}_1 \times$ **Env** $\longrightarrow \mathbb{B}$
Index	:	**kType** $\times \mathbb{N}_1 \longrightarrow$ **kType**
Flatten	:	**kType** $\longrightarrow$ **kTypeSeq**
Not-Local-Type	:	**kType** $\times$ **Env** $\longrightarrow \mathbb{B}$
TypeTuple	:	**kTypeSeq** $\longrightarrow$ **kType**
ConstTuple	:	**kConstSeq** $\longrightarrow$ **kConst**
ConstsetTuple	:	**kConstsetSeq** $\longrightarrow$ **kConstset**
UnitTuple	:	**kUnitSeq** $\longrightarrow$ **kUnit**
Disjoint	:	**kConstset** $\times$ **kConstset** $\longrightarrow \mathbb{B}$

$\_ \vdash \_ \;\boxed{\text{EM}}\Rightarrow \_$	$\subseteq$	*Env* $\times$ *Enum* $\times$ *kEnum*
$\_ \vdash \_ \;\boxed{\text{T}}\Rightarrow \_$	$\subseteq$	*Env* $\times$ *Type* $\times$ *kType*
$\_ \vdash \_ \;\boxed{\text{C}}\Rightarrow \_ : \_$	$\subseteq$	*Env* $\times$ *Const* $\times$ *kConst* $\times$ *kType*
$\_ \vdash \_ \;\boxed{\text{CS}}\Rightarrow \_ : \_$	$\subseteq$	*Env* $\times$ *Constset* $\times$ *kConstset* $\times$ *kType*
$\_ \vdash \_ \;\boxed{\text{U}}\Rightarrow \_ : \_ , \_$	$\subseteq$	*Env* $\times$ *Unit* $\times$ *kUnit* $\times$ *kType* $\times$ *Env*
$\_ \vdash \_ \;\boxed{\text{CC}}\Rightarrow \_ : \_ , \_$	$\subseteq$	*Env* $\times$ *Closedclause* $\times$ *kUnit* $\times$ *kType* $\times$ *Env*
$\_ \vdash \_ \;\boxed{\text{CA}}\Rightarrow \_ , \_ : \_ , \_ , \_$	$\subseteq$	*Env* $\times$ *Case* $\times$ *kConstset* $\times$ *kUnit* $\times$ *kType* $\times$ *kType* $\times$ *Env*
$\_ \vdash \_ \;\boxed{\text{SP}}\Rightarrow \_$	$\subseteq$	*Env* $\times$ *Step* $\times$ *Env*
$\_ \vdash \_ \;\boxed{\text{NW}}\Rightarrow \_ , \_$	$\subseteq$	*Env* $\times$ *New* $\times$ *kNew* $\times$ *Env*
$\_ \vdash \_ \;\boxed{\text{TA}}\Rightarrow \_ , \_$	$\subseteq$	*Env* $\times$ *Typealt* $\times$ *Tag* $\times$ *Env*
$\_ \vdash \_ \;\boxed{\text{TD}}\Rightarrow \_$	$\subseteq$	*Env* $\times$ *Typedec* $\times$ *Env*
$\_ \vdash \_ \;\boxed{\text{FD}}\Rightarrow \_$	$\subseteq$	*Env* $\times$ *Fndec* $\times$ *Env*
$\_ \vdash \_ \{\_ , \_\} \;\boxed{\text{BI}}\Rightarrow \_$	$\subseteq$	*Env* $\times$ *Builtin* $\times$ *kType* $\times$ *kType* $\times$ *kBuiltin*
$\_ \vdash \_ \;\boxed{\text{IN}}\Rightarrow \_ , \_$	$\subseteq$	*Env* $\times$ *Inputfnspec* $\times$ *kType* $\times$ *Env*
$\_ \vdash \_ \;\boxed{\text{D}}\Rightarrow \_$	$\subseteq$	*Env* $\times$ *Declaration* $\times$ *Env*
$\_ \vdash \_ \;\boxed{\text{CL}}\Rightarrow \_$	$\subseteq$	*Env* $\times$ *Closure* $\times$ *Env*
$\_ \;\boxed{\text{KERNEL}}\Rightarrow \_$	$\subseteq$	*Closure* $\times$ *kClosure*
$\_ \;\boxed{\text{T}}= \_$	$\subseteq$	*kType* $\times$ *kType* $\rightarrow \mathbb{B}$

3.4.4 Scopes

The scopes of Core ELLA are removed by the transformation to the Kernel. In order to achieve this the following are needed

Scope-Fn-Begin*: *Env* $\rightarrow$ *Env

***Scope-Fn-Begin*(*E*)** $\underline{\triangle}$

$$
\begin{array}{llll}
\textbf{env}(& E.typedec, & E.fndec, [\,], & \\
 & E.fnmap \dagger E.lclfnmap, & \{\,\}, & E.tynamemap \dagger E.lcltynamemap, \\
 & \{\,\}, & \{\,\}, & \{\,\} \\
 & \{\,\} & \{\,\} & \{\,\})
\end{array}
$$

$$\textit{Scope-Fn-End}: Env \times Env \rightarrow Env$$

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textit{Scope-Fn-End}(E, L) \triangleq} \\
\mathbf{env}(& L.typedec, & L.fndec, & E.sigdec, \\
 & E.fnmap, & E.lclfnmap \dagger L.lclfnmap, & \\
 & E.tynamemap, & E.lcltynamemap \dagger L.lcltynamemap, & \\
 & E.signamemap, & E.lclsignamemap, & \\
 & E.usedtyname, & E.usedfnname & E.usedsigname)
\end{array}
$$

$$\textit{Scope-Begin}: Env \rightarrow Env$$

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textit{Scope-Begin}(E) \triangleq} \\
\mathbf{env}(& E.typedec, & E.fndec, & E.sigdec, \\
 & E.fnmap \dagger E.lclfnmap, & \{\}, & \\
 & E.tynamemap \dagger E.lcltynamemap, & \{\}, & \\
 & E.signamemap \dagger E.lclsignamemap, & \{\} & \\
 & E.usedtyname & E.usedfnname & E.usedsigname)
\end{array}
$$

$$\textit{Scope-End}: Env \times Env \rightarrow Env$$

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textit{Scope-End}(E, L) \triangleq} \\
\mathbf{env}(& L.typedec, & L.fndec, & L.sigdec, \\
 & E.fnmap, & E.lclfnmap, & \\
 & E.tynamemap, & E.lcltynamemap, & \\
 & E.signamemap, & E.lclsignamemap, & \\
 & E.usedtyname \dagger \mathsf{dom}\,(L.lcltynamemap) & E.usedfnname \dagger \mathsf{dom}\,(L.lclfnmap) & \\
 & E.usedsigname \dagger \mathsf{dom}\,(L.lclsignamemap)) & &
\end{array}
$$

The stacking and unstacking of the scopes for BEGIN..END clauses is carried out through the transformation rule [CC3]. Whilst the stacking and unstacking of local function and type declarations are carried out through the transformation rules [SP1] and [SP2].

3.4.5 Join Checks

The *Check-Joins* predicate is used to ensure that all local signals in an Environment have been joined.

$$\textit{Check-Joins} : Env \rightarrow \mathbb{B}$$

$$\textit{Check-Joins}(E) \;\triangleq\; \forall s \in \mathsf{rng}\, E.lclsignamemap \cdot s.sort = \mathbf{joined}$$

3.4.6 Two Value Types

Here we present the predicate for checking that a type is a two valued enumerated type:

$Check\text{-}Two\text{-}Val : kType \to \mathbf{B}$

$Check\text{-}Two\text{-}Val(ty) \ \underline{\triangle}$
$\quad \mathsf{let}\ \mathbf{typeno}(typeno) = ty\ \mathsf{in}$
$\quad \mathsf{let}\ (E.typedec)[typeno].new = \mathbf{tags}(TagSeq)\ \mathsf{in}$
$\quad\quad \mathsf{len}(\mathbf{tags}(TagSeq)) = 2$

3.4.7 Check names

These predicates ensure that a particular name is not already in scope. They will be used by the functions that add names to an Environment.

$Check\text{-}Fn: Fnname \times Env \to \mathbf{B}$

$Check\text{-}Fn(fnname, E) \ \underline{\triangle}$
$\quad fnname \notin (\mathsf{dom}(E.lclfnmap) \cup E.usedfnname)$

$Check\text{-}Typename: Name \times Env \to \mathbf{B}$

$Check\text{-}Typename(name, E) \ \underline{\triangle}$
$\quad name \notin (\mathsf{dom}(E.lcltynamemap) \cup E.usedtyname)$

$Check\text{-}Signal: Signalname \times Env \to \mathbf{B}$

$Check\text{-}Signal(signalname, E) \ \underline{\triangle}$
$\quad signalname \notin (\mathsf{dom}(E.signamemap) \cup \mathsf{dom}(E.lclsignamemap) \cup E.usedsigname)$

3.4.8 Adding Names to an Environment

These functions define the addition of names to an environment

$Add\text{-}Fn: Fndec \times Env \times Env \to Env$

$Add\text{-}Fn(fd, E_1, E_2) \ \underline{\triangle}$
$\quad \mathsf{let}\ E = \mu(E_1, \{typedec \mapsto E_2.typedec,\ fndec \mapsto E_2.fndec\})\ \mathsf{in}$
$\quad \mathsf{let}\ Len = \mathsf{len}\ E.fndec\ \mathsf{in}$
$\quad \mathsf{let}\ FnName = fd.fnname\ \mathsf{in}$
$\quad\quad \mu(E, \{ fndec \mapsto E.fndec \frown [fd],$
$\quad\quad\quad lclfnmap \mapsto (E.lclfnmap \dagger \{FnName \mapsto Len + 1\})$
$\quad\quad\quad \}$
$\quad\quad)$
pre
$\quad Check\text{-}Fn(fd.fnname, E_1)$

$Add\text{-}Type\colon Typedec \times Env \to Env$

$Add\text{-}Type(td, E) \ \underline{\triangle}$
    let $Len = \mathsf{len}\ E.typedec$ in
    let $TyName = td.typename$ in
$$\mu(E, \{\ typedec \mapsto E.typedec \curvearrowright [td],\ lcltynamemap \mapsto (E.lcltynamemap \dagger \{TyName \mapsto \ \mathbf{typeno}(Len + 1)\})\ \})$$
pre
$$Check\text{-}Typename(td.typename, E) \wedge Check\text{-}Signal(td.typename, E)$$

$Add\text{-}Signal\colon Signaldec \times Sort \times Env \to Env$

$Add\text{-}Signal(sd, sort, E) \ \underline{\triangle}$
    let $Len = \mathsf{len}\ E.sigdec$ in
    let $SigName = sd.signalname$ in
$$\mu(E, \{\ sigdec \mapsto E.sigdec \curvearrowright [sd],\ lclsignamemap \mapsto (E.lclsignamemap \dagger \{SigName \mapsto \mathbf{sig}(Len + 1, sort)\})\ \})$$
pre
$$Check\text{-}Signal(sd.signalname, E) \wedge Check\text{-}Typename(sd.signalname, E)$$

$Add\text{-}Join\colon Signaldec \times Signalno \times Env \to Env$

$Add\text{-}Join(sd, signalno, E) \ \underline{\triangle}$
    let $SigName = sd.signalname$ in
$$\mu(E, \{\ sigdec[signalno] \mapsto sd,\ lclsignamemap \mapsto (E.lclsignamemap \dagger \{SigName \mapsto \mathbf{sig}(signalno,\ \mathbf{joined})\})\ \})$$

$Add\text{-}Tag\colon Tagname \times Env \to Env$

$Add\text{-}Tag(tagname, E) \ \underline{\triangle}$
    let $Len = \mathsf{len}\ E.typedec$ in
$$\mu(E, lcltynamemap \mapsto (E.lcltynamemap \dagger \{tagname \mapsto \ \mathbf{consttag}(Len + 1)\}))$$
pre
$$Check\text{-}Typename(tagname, E) \wedge Check\text{-}Signal(tagname, E)$$

$Add\text{-}Type\text{-}Name\colon Typename \times kType \times Env \to Env$

$Add\text{-}Type\text{-}Name(typename, ktype, E) \ \underline{\triangle}$
$$\mu(E, lcltynamemap \mapsto (E.lcltynamemap \dagger \{typename \mapsto \ \mathbf{typename}(typename, ktype)\}))$$
pre
$$Check\text{-}Typename(typename, E) \wedge Check\text{-}Signal(typename, E)$$

3.4.9 Finding Names in an Environment

These functions describe how any name can be located within an Environment

$\textit{Find-Type-Nm}: \textit{Name} \times \textit{Env} \rightarrow \textit{Typetag}$

$\textit{Find-Type-Nm}(\textit{name}, E) \triangleq$
$\quad (E.\textit{tynamemap} \dagger E.\textit{lcltynamemap})(\textit{name})$

$\textit{Find-Sig-Nm}: \textit{Name} \times \textit{Env} \rightarrow \textit{Sig}$

$\textit{Find-Sig-Nm}(\textit{signalname}, E) \triangleq$
$\quad (E.\textit{signamemap} \dagger E.\textit{lclsignamemap})(\textit{signalname})$

$\textit{Find-Fn}: \textit{Fnname} \times \textit{Env} \rightarrow \textit{Fnno}$

$\textit{Find-Fn}(\textit{name}, E) \triangleq$
$\quad (E.\textit{fnmap} \dagger E.\textit{lclfnmap})(\textit{name})$

$\textit{Find-Unjoined}: \textit{Signalname} \times \textit{Env} \rightarrow \textit{Fnno}$

$\textit{Find-Unjoined}(\textit{signalname}, E) \triangleq$
$\quad \mathsf{let}\ \textit{Signo} = (E.\textit{lclsignamemap})(\textit{signalname}).\textit{signalno}\ \mathsf{in}$
$\quad \mathsf{let}\ \mathbf{signaldec}(\textit{signalname}, \textit{type}, \mathbf{instance}(\textit{fnno}, \_)) = (E.\textit{sigdec})[\textit{Signo}]\ \mathsf{in}$
$\qquad \textit{fnno}$
pre
$\quad (E.\textit{lclsignamemap})(\textit{signalname}).\textit{sort} = \mathbf{unjoined}$

$\textit{Find-Type}: \textit{Typename} \times \textit{Env} \rightarrow \textit{kType}$

$\textit{Find-Type}(\textit{typename}, E) \triangleq$
$\quad \textit{Find-Type-Nm}(\textit{typename}, E)$
pre
$\quad \textit{Find-Type-Nm}(\textit{typename}, E) \in (\mathbf{typename} \cup \mathbf{typeno})$

$\textit{Find-Alt}: \textit{Altname} \times \textit{Env} \rightarrow \textit{kEnum}$

$\textit{Find-Alt}(\textit{altname}, E) \triangleq$
$\quad \mathsf{let}\ \mathbf{consttag}(\textit{ktypeno}) = \textit{Find-Type-Nm}(\textit{altname}, E)\ \mathsf{in}$
$\quad \mathsf{let}\ \mathbf{typedec}(\_, \mathbf{tags}(\textit{tags})) = (E.\textit{typedec})[\textit{ktypeno}]\ \mathsf{in}$
$\quad \mathsf{let}\ \textit{index} = \iota\,(i \in \mathsf{inds}\ \textit{tags}) \cdot \textit{tags}[i] = \mathbf{tag}(\textit{altname}, \_)\ \mathsf{in}$
$\qquad \mathbf{enum}(\textit{ktypeno}, \textit{index})$

Find-ELLAint: *Tagname* × *Env* → *Typeno* × *Lowerbound* × *Upperbound*

Find-ELLAint(*tagname*, *E*) $\underline{\triangle}$
  let **consttag**(*ktypeno*) = *Find-Type-Nm*(*tagname*, *E*) in
  let **typedec**(_, **ellaint**(_, *lb*, *ub*)) = (*E*.*typedec*)[*ktypeno*] in
    ktypeno, *lb*, *ub*

Find-Integer-Type: *kType* × *Env* → *Typeno* × *Lowerbound* × *Upperbound*

Find-Integer-Type(*ktype*, *E*) $\underline{\triangle}$
  cases *ktype* of
   typeno(*typeno*) → cases (*E*.*typedec*)[*typeno*] of
        typedec(_, **ellaint**(*t*, *l*, *u*)) → *t*, *l*, *u*
        end
   typename(_, *type*) → *Find-Integer-Type*(*type*)
  end

Find-Char: *Tagname* × *Char* × *Env* → *kEnum*

Find-Char(*tagname*, *char*, *E*) $\underline{\triangle}$
  let **consttag**(*ktypeno*) = *Find-Type-Nm*(*tagname*, *E*) in
  let **typedec**(_, **chars**(_, *chs*)) = (*E*.*typedec*)[*ktypeno*] in
  let *index* = ι(*i* ∈ inds *chs*) · *chs*[*i*] = *char* in
    enum(*ktypeno*, *index*)

Find-Signal: *Signal* × *Env* → *kUnit* × *kType*

Find-Signal(*signalname*, *E*) $\underline{\triangle}$
  let **sig**(*signalno*, _) = *Find-Sig-Nm*(*signalname*, *E*) in
    signal(*signalno*), (*E*.*sigdec*)[*signalno*].*type*

Find-Assoc: *Altname* × *Env* → *kType*

Find-Assoc(*altname*, *E*) $\underline{\triangle}$
  let **consttag**(*ktypeno*) = *Find-Type-Nm*(*altname*, *E*) in
  let **typedec**(_, **tags**(*tags*)) = (*E*.*typedec*)[*ktypeno*] in
   ι(*ktypeOpt* ∈ *kTypeOpt*) · **tag**(*altname*, *ktypeOpt*) ∈ elems *tags*

$$\textit{Find-Row}: \textit{kType} \rightarrow \mathbb{N}_1 \times \textit{kType}$$

Find-Row(*ktype*) $\underline{\triangle}$
  let **types**(*types*) = *ktype* in
  let *size* = len (*types*) in
    (*size*, *types*[1])
pre
  $types[1] \dashv\boxed{T}\vDash types[i] \qquad \textit{ForAll } i \in \{1..size\}$

$$\textit{Find-Biop}: \textit{Biopname} \times \textit{kType} \times \textit{kType} \rightarrow \textit{kFnbody}$$

Find-Biop(*name*, $ktype_1$, $ktype_2$) $\underline{\triangle}$
  let *biopinfo* = $\iota(i \in BiopEnv.biop) \cdot i.biopname = name$ in
    biop(*name*)
pre
  $(biopinfo.inputtype \dashv\boxed{T}\vDash ktype_1) \wedge (biopinfo.outputtype \dashv\boxed{T}\vDash ktype_2)$

3.4.10 Removing Type Aliasing

Type aliasing is removed by means of the following function:

$$\textit{Get-Type} : \textit{kType} \rightarrow \textit{kType}$$

Get-Type(*ty*) $\underline{\triangle}$
  cases *ty* of
  types([$ktype_1, \cdots, ktype_k$]) → **types**([*Get-Type*($ktype_1$), ⋯, *Get-Type*($ktype_k$)]),
  typename(_, *ktype*) → *Get-Type*(*ktype*)
  stringtype(*size*, *ktype*) → **stringtype**(*size*, *Get-Type*(*ktype*))
  others *ty*

  end

3.4.11 Type Checking

Type checking is an important aspect of the ELLA compiler and the relation '$a \dashv\boxed{T}\vDash b$' shows how the transformation from Core ELLA to the **Kernel** will define type equality. This relation is defined by:-

$$ktype_1 \dashv\boxed{T}\vDash ktype_2 \Leftrightarrow \textit{Type-Equals}(ktype_1, ktype_2)$$

where

$$\textit{Type-Equals} : kType \times kType \rightarrow \mathbf{B}$$

$$\begin{array}{l}
\textit{Type-Equals}(ty_1, ty_2) \;\triangleq \\
\quad \textsf{cases}\ (\textit{Get-Type}(ty_1), \textit{Get-Type}(ty_2))\ \textsf{of} \\
\quad (\ \mathbf{typeno}(typeno_1),\ \mathbf{typeno}(typeno_2)) \rightarrow (typeno_1 = typeno_2) \\
\quad (\ \mathbf{stringtype}(s_1, tn_1),\ \mathbf{stringtype}(s_2, tn_2)) \rightarrow (s_1 = s_2) \wedge \textit{Type-Equals}(tn_1, tn_2) \\
\quad (\ \mathbf{types}([t_1, ..., t_k]),\ \mathbf{types}([s_1, ..., s_j])) \rightarrow j = k \bigwedge_{i=1..k} \textit{Type-Equals}(t_i,\ s_i) \\
\quad (\ \mathbf{typevoid},\ \mathbf{typevoid}) \rightarrow \textsf{true} \\
\quad \textsf{others false} \\
\quad \textsf{end}
\end{array}$$

3.4.12 Concatenation

Concatenation of two signal types are handled by means of the following function.

$$Conc : kType \times kType \rightarrow kType$$

$$\begin{array}{l}
Conc(ty_1, ty_2) \;\triangleq \\
\quad \textsf{cases}\ (Get\text{-}Type(ty_1), Get\text{-}Type(ty_2))\ \textsf{of} \\
\quad (\ \mathbf{types}([ta_1, \cdots, ta_k]), \\
\qquad \mathbf{types}([tb_1, \cdots, tb_l])) \rightarrow \textsf{let}\ (\forall i \in \{1..k\}, j \in \{1..l\} \cdot (ta_i =\!\boxed{T}\!= tb_j)) = \textsf{true in} \\
\qquad\qquad \mathbf{types}([ta_1, \cdots, ta_k, tb_1, \cdots, tb_l]) \\
\quad (\ \mathbf{types}([t_1, \cdots, t_k]),\ _) \rightarrow \textsf{let}\ (\forall i \in \{1..k\} \cdot t_i =\!\boxed{T}\!= ty_2) = \textsf{true in} \\
\qquad\qquad \mathbf{types}([t_1, \cdots, t_k, ty_2]) \\
\quad (_,\ \ \mathbf{types}([t_1, \cdots, t_k])) \rightarrow \textsf{let}\ (\forall i \in \{1..k\} \cdot t_i =\!\boxed{T}\!= ty_1) = \textsf{true in} \\
\qquad\qquad \mathbf{types}([ty_1, t_1, \cdots, t_k]) \\
\quad (\ \mathbf{stringtype}(size_a, ktype_a), \\
\qquad \mathbf{stringtype}(size_b, ktype_b)) \rightarrow \textsf{let}\ (ktype_a =\!\boxed{T}\!= ktype_b) = \textsf{true in} \\
\qquad\qquad \mathbf{stringtype}(size_a + size_b, ktype_a) \\
\quad (\ \mathbf{stringtype}(size, ktype),\ _) \rightarrow \textsf{let}\ (ktype =\!\boxed{T}\!= ty_2) = \textsf{true in} \\
\qquad\qquad \mathbf{stringtype}(size + 1, ktype) \\
\quad (_,\ \ \mathbf{stringtype}(size, ktype)) \rightarrow \textsf{let}\ (ty_1 =\!\boxed{T}\!= ktype) = \textsf{true in} \\
\qquad\qquad \mathbf{stringtype}(size + 1, ktype) \\
\quad \textsf{end}
\end{array}$$

3.4.13 Character Check

This rule checks that a particular type is of the form of an ELLA character.

$$\textit{Is-Char} : \mathbb{N}_1 \times Env \rightarrow \mathbf{B}$$

$$\textit{Is-Char}(typeno, E) \;\triangleq\; (E.typedec)[typeno].new \in \mathbf{chars}$$

3.4.14 Type Indexing

This function describes how to obtain the type of an indexed type

$Index : kType \times \mathbb{N}_1 \rightarrow kType$

$Index(ty, i) \triangleq$
cases $Get\text{-}Type(ty)$ **of**
$\mathbf{types}([ktype_1, \cdots, ktype_k]) \rightarrow ktype_i,$
$\mathbf{stringtype}(\_, ktype) \rightarrow ktype$
end

3.4.15 Reform

This function flattens types so that they are available for **reform**

$Flatten : kType \rightarrow kTypeSeq$

$Flatten(ty) \triangleq$
cases $Get\text{-}Type(ty)$ **of**
$\mathbf{typeno}\ (typeno) \rightarrow [\ \mathbf{typeno}(typeno)],$
$\mathbf{stringtype}\ (size, ktype) \rightarrow [\ \mathbf{stringtype}(size, ktype)],$
$\mathbf{types}\ ([t_1, \cdots, t_k]) \rightarrow Flatten(t_1) ^\frown \cdots ^\frown Flatten(t_k)$
$\mathbf{typevoid} \rightarrow [\ \mathbf{typevoid}]$
end

3.4.16 Local Type Checking

This function checks that its input type is not a locally declared type, and will be used by local BEGIN..END clauses to ensure that the output from the clause only contains global types.

$Not\text{-}Local\text{-}Type : kType \times Env \rightarrow \mathbb{B}$

$Not\text{-}Local\text{-}Type(ktype, E) \triangleq$
cases $Get\text{-}Type(ktype)$ **of**
$\mathbf{typeno}(typeno) \rightarrow \forall (i \in \mathrm{rng}\ E.lcltynamemap) \cdot i \neq \mathbf{typeno}(typeno)$
$\mathbf{types}([t_1, \cdots, t_n]) \rightarrow \bigwedge_{i \in \{1..n\}} Not\text{-}Local\text{-}Type(t_i, E)$
$\mathbf{stringtype}(size, t) \rightarrow Not\text{-}Local\text{-}Type(t, E)$
others true
end

3.4.17 Constructing Tuples

These functions convert sequences into tuples.

$TypeTuple : kTypeSeq \rightarrow kType$

$$TypeTuple(tseq) \triangleq \begin{array}{l} \textsf{if len } tseq = 1 \\ \textsf{then } tseq[1] \\ \textsf{else } \ \mathbf{types}\ (tseq) \end{array}$$

$ConstTuple : kConstSeq \rightarrow kConst$

$$ConstTuple(cseq) \triangleq \begin{array}{l} \textsf{if len } cseq = 1 \\ \textsf{then } cseq[1] \\ \textsf{else } \ \mathbf{consts}\ (cseq) \end{array}$$

$ConstsetTuple : kConstsetSeq \rightarrow kConstset$

$$ConstsetTuple(csseq) \triangleq \begin{array}{l} \textsf{if len } csseq = 1 \\ \textsf{then } csseq[1] \\ \textsf{else } \ \mathbf{constsets}\ (csseq) \end{array}$$

$UnitTuple : kUnitSeq \rightarrow kUnit$

$$UnitTuple(useq) \triangleq \begin{array}{l} \textsf{if len } useq = 1 \\ \textsf{then } useq[1] \\ \textsf{else } \ \mathbf{units}\ (useq) \end{array}$$

3.4.18 Case Disjointness

Within CASE statements all the different arms must have distinctive choosers. In order to ensure this the following is needed.

$$Disjoint : kConstset \times kConstset \rightarrow \mathbb{B}$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{Disjoint(cset_1, cset_2) \triangleq} \\
\quad \textsf{cases}\ (cset_1, cset_2)\ \textsf{of} & \\
\quad (\ \mathbf{enum}(\_, tagno_1),\ \ \mathbf{enum}(\_, tagno_2)) & \rightarrow tagno_1 \neq tagno_2 \\
\quad (\ \mathbf{string}(\_, [tagno_{11}, \cdots, tagno_{1k}]), & \\
\quad\quad \mathbf{string}(\_, [tagno_{21}, \cdots, tagno_{2k}])) & \rightarrow \bigvee_{i=\{1..k\}} (tagno_{1i} \neq tagno_{2i}) \\
\quad (\ \mathbf{constsetassoc}(\ \mathbf{enum}(\_, tagno_1), constset_1), & \\
\quad\quad \mathbf{constsetassoc}(\ \mathbf{enum}(\_, tagno_2), constset_2)) & \rightarrow tagno_1 \neq tagno_2 \vee \\
 & \quad Disjoint(constset_1,\ constset_2) \\
\quad (\ \mathbf{constsets}([csa_1, \cdots, csa_k]), & \\
\quad\quad \mathbf{constsets}([csb_1, \cdots, csb_k])) & \rightarrow \bigvee_{i=\{1..k\}} Disjoint(csa_i,\ csb_i) \\
\quad (enum,\ \ \mathbf{constsetalts}([csa_1, \cdots, csa_k])) & \rightarrow \bigwedge_{i=\{1..k\}} Disjoint(enum,\ csa_i) \\
\quad (constsetalts,\ enum) & \rightarrow Disjoint(enum,\ constsetalts) \\
\quad (\ \mathbf{constsetalts}([csa_1, \cdots, csa_k]), & \\
\quad\quad constsetalts) & \rightarrow \bigwedge_{i=\{1..k\}} Disjoint(csa_i,\ constsetalts) \\
\quad (\ \mathbf{constsetstring}(size_a, cset_a), & \\
\quad\quad \mathbf{constsetstring}(size_b, cset_b)) & \rightarrow (size_a \neq size_b) \vee \\
 & \quad Disjoint(cset_a,\ cset_b) \\
\quad (\ \mathbf{constsetany}(type),\ \_) & \rightarrow \textsf{false} \\
\quad (\_,\ \ \mathbf{constsetany}(type)) & \rightarrow \textsf{false} \\
\quad \textsf{end} &
\end{array}
$$

3.5 Formal Transformation System

This section describes transformations from Core ELLA to the **Kernel**. These include the semantic checks which are done by the full ELLA compiler on Core ELLA ie. type checking, name checking etc. Thus this section includes a description of the static semantics of Core ELLA. At the start of each subsection the Core ELLA syntax, for which the transformations of that section apply, will be given.

3.5.1 Enumerated Values

Enumerated values are defined by

enumerated	::=	altname
	|	tagname / z
	|	tagname 'char
	|	tagname "string"

and the transformations on them are given by

$$\boxed{\textsf{EM1}}\ \frac{\textit{Find-Alt}\ (altname, E) =\ \mathbf{enum}(ktypeno, index)}{E \vdash [\![altname]\!] \overset{\boxed{\text{EM}}}{\Longrightarrow}\ \mathbf{enum}(ktypeno, index)}$$

$$\boxed{\text{EM2}}\;\frac{\textit{Find-ELLAint}\ (tagname, E)\ =\ ktypeno,\ lb,\ ub \qquad lb \leq z \leq ub}{E \vdash [\![tagname/z]\!] =\!\boxed{\text{EM}}\!\Rightarrow\ \textbf{enum}(ktypeno, z\text{-}lb+1)}$$

$$\boxed{\text{EM3}}\;\frac{\textit{Find-Char}\ (tagname, char, E) =\ \textbf{enum}(ktypeno, index)}{E \vdash [\![tagname\ 'char]\!] =\!\boxed{\text{EM}}\!\Rightarrow\ \textbf{enum}(ktypeno, index)}$$

$$\boxed{\text{EM4}}\;\frac{\textit{Find-Char}\ (tagname, char_i, E)\ =\ \textbf{enum}(ktypeno, ktagno_i) \qquad \textit{ForAll}\ i \in \{1..k\}}{E \vdash [\![tagname\ "char_1 \cdots char_k"]\!] =\!\boxed{\text{EM}}\!\Rightarrow\ \textbf{string}(ktypeno, [ktagno_1, \cdots, ktagno_k])}$$

3.5.2 Types

Types in Core ELLA can have the following form

type	::=	typename
	|	STRING [size] typename
	|	[size] type
	|	($\text{type}_1, \cdots, \text{type}_k$)
	|	()

and the transformations that apply to them are

$$\boxed{\text{T1}}\;\frac{\textit{Find-Type}\ (typename, E) = ktype}{E \vdash [\![typename]\!] =\!\boxed{\text{T}}\!\Rightarrow\ ktype}$$

$$\boxed{\text{T2}}\;\frac{E \vdash [\![typename]\!] =\!\boxed{\text{T}}\!\Rightarrow\ ktype \qquad \textit{Get-Type}(ktype) =\ \textbf{typeno}(ktypeno) \qquad \textit{Is-Char}\ (ktypeno)}{E \vdash [\![STRING[size]typename]\!] =\!\boxed{\text{T}}\!\Rightarrow\ \textbf{stringtype}(size, ktype)}$$

$$\boxed{\text{T3}}\;\frac{E \vdash [\![type]\!] =\!\boxed{\text{T}}\!\Rightarrow\ ktype}{E \vdash [\![[size]type]\!] =\!\boxed{\text{T}}\!\Rightarrow\ \textbf{types}([ktype^{size}])}$$

$$\boxed{\text{T4}}\;\frac{E \vdash [\![type_i]\!] =\!\boxed{\text{T}}\!\Rightarrow\ t_i \qquad \textit{ForAll}\ i \in \{1..k\}}{E \vdash [\![(type_1, \cdots, type_k)]\!] =\!\boxed{\text{T}}\!\Rightarrow\ \textit{TypeTuple}([t_1, \cdots, t_k])}$$

$$\boxed{\text{T5}}\;\frac{}{E \vdash [\![\ ()\ ]\!] =\!\boxed{\text{T}}\!\Rightarrow\ \textbf{typevoid}}$$

3.5.3 Constants

The Core ELLA definition of constants is

const	::=	STRING [size] const1
	|	[size] const
	|	const1
const1	::=	enumerated
	|	altname & const1
	|	($\text{const}_1, \cdots, \text{const}_k$)
	|	? type
	|	()

with their transformation rules being

$$\boxed{\text{C1}}\ \frac{E \vdash [\![\mathit{const1}]\!] \dashv\!\boxed{\text{C}}\!\Rightarrow \mathit{kconst}\text{: } \mathbf{typeno}(\mathit{ktypeno}) \qquad \mathit{Is\text{-}Char}\ (\mathit{ktypeno},\ E)}{E \vdash [\![\mathit{STRING}[\mathit{size}]\mathit{const1}]\!] \dashv\!\boxed{\text{C}}\!\Rightarrow \mathbf{conststring}(\mathit{size}, \mathit{kconst})\text{: } \mathbf{stringtype}(\mathit{size},\ \mathbf{typeno}(\mathit{ktypeno}))}$$

$$\boxed{\text{C2}}\ \frac{E \vdash [\![\mathit{const}]\!] \dashv\!\boxed{\text{C}}\!\Rightarrow \mathit{kconst}\text{: } \mathit{ktype}}{E \vdash [\![[\mathit{size}]\mathit{const}]\!] \dashv\!\boxed{\text{C}}\!\Rightarrow \mathbf{consts}([\mathit{kconst}^{\mathit{size}}])\text{: } \mathbf{types}([\mathit{ktype}^{\mathit{size}}])}$$

$$\boxed{\text{C3}}\ \frac{E \vdash [\![\mathit{enumerated}]\!] \dashv\!\boxed{\text{EM}}\!\Rightarrow \mathbf{enum}(\mathit{ktypeno}, \mathit{tagno})}{E \vdash [\![\mathit{enumerated}]\!] \dashv\!\boxed{\text{C}}\!\Rightarrow \mathbf{enum}(\mathit{ktypeno}, \mathit{tagno})\text{: } \mathbf{typeno}(\mathit{ktypeno})}$$

$$\boxed{\text{C4}}\ \frac{E \vdash [\![\mathit{enumerated}]\!] \dashv\!\boxed{\text{EM}}\!\Rightarrow \mathbf{string}(\mathit{ktypeno}, \mathit{tagnoseq})}{E \vdash [\![\mathit{enumerated}]\!] \dashv\!\boxed{\text{C}}\!\Rightarrow \mathbf{string}(\mathit{ktypeno}, \mathit{tagnoseq})\text{: } \mathbf{typeno}(\mathit{ktypeno})}$$

$$\boxed{\text{C5}}\ \frac{\begin{array}{c} E \vdash [\![\mathit{const1}]\!] \dashv\!\boxed{\text{C}}\!\Rightarrow \mathit{kconst}\text{: } \mathit{ktype}_1 \\ \mathit{Find\text{-}Assoc}\ (\mathit{altname}, E)\ =\ \mathit{ktype}_2 \\ \mathit{ktype}_1 \dashv\!\boxed{\text{T}}\!\vDash \mathit{ktype}_2 \qquad \mathit{Find\text{-}Alt}\ (\mathit{altname}, E)\ =\ \mathbf{enum}(\mathit{ktypeno}, \mathit{index}) \end{array}}{E \vdash [\![\mathit{altname}\&\mathit{const1}]\!] \dashv\!\boxed{\text{C}}\!\Rightarrow \mathbf{constassoc}(\ \mathbf{enum}(\mathit{ktypeno}, \mathit{index}), \mathit{kconst})\text{: } \mathbf{typeno}(\mathit{ktypeno})}$$

$$\boxed{\text{C6}}\ \frac{E \vdash [\![\mathit{const}_i]\!] \dashv\!\boxed{\text{C}}\!\Rightarrow \mathit{kconst}_i\text{: } \mathit{ktype}_i \qquad \mathit{ForAll}\ i \in \{1..k\}}{E \vdash [\![(\mathit{const}_1, \cdots, \mathit{const}_k)]\!] \dashv\!\boxed{\text{C}}\!\Rightarrow \mathit{ConstTuple}([\mathit{kconst}_1, \cdots, \mathit{kconst}_k])\text{: } \mathit{TypeTuple}([\mathit{ktype}_1, \cdots, \mathit{ktype}_k])}$$

$$\boxed{\text{C7}}\ \frac{E \vdash [\![\mathit{type}]\!] \ \boxed{\text{T}} \Rightarrow \ \mathit{ktype}}{E \vdash [\![?\mathit{type}]\!] \ \boxed{\text{C}} \Rightarrow \ \textbf{constquery}(\mathit{ktype}) \colon \mathit{ktype}}$$

$$\boxed{\text{C8}}\ \frac{}{E \vdash [\![()]\!] \ \boxed{\text{C}} \Rightarrow \ \textbf{constvoid} \colon \textbf{typevoid}}$$

3.5.4 Constant Sets

Constant sets are given by

constset	::=	constset_1 | $\cdots$ | constset_k
constset1	::=	STRING [size] constset2
	|	[size] constset1
	|	constset2
constset2	::=	enumerated
	|	altname & constset2
	|	($\text{constset}_1, \cdots, \text{constset}_k$)
	|	type

with transformations on them given by

$$\boxed{\text{CS1}}\ \frac{E \vdash [\![\mathit{constset}_i]\!] \ \boxed{\text{CS}} \Rightarrow \ \mathit{kcset}_i \colon \mathit{ktype}_i \qquad \mathit{ktype}_i \ \boxed{\text{T}} = \ \mathit{ktype}_1 \qquad \mathit{ForAll}\ i \in \{1..k\}}{E \vdash [\![\mathit{constset}_1 \mid \cdots \mid \mathit{constset}_k]\!] \ \boxed{\text{CS}} \Rightarrow \ \textbf{constsetalts}([\mathit{kcset}_1, \cdots, \mathit{kcset}_k]) \colon \mathit{ktype}_1}$$

$$\boxed{\text{CS2}}\ \frac{\begin{array}{c} E \vdash [\![\mathit{constset2}]\!] \ \boxed{\text{CS}} \Rightarrow \ \mathit{kcset} \colon \textbf{typeno}(\mathit{ktypeno}) \\ \mathit{Is\text{-}Char}\ (\mathit{ktypeno},\ E) \end{array}}{\begin{array}{c} E \vdash [\![\mathit{STRING}[\mathit{size}]\mathit{constset2}]\!] \ \boxed{\text{CS}} \Rightarrow \\ \textbf{constsetstring}(\mathit{size}, \mathit{kcset}) \colon \textbf{stringtype}(\mathit{size},\ \textbf{typeno}(\mathit{ktypeno})) \end{array}}$$

$$\boxed{\text{CS3}}\ \frac{E \vdash [\![\mathit{constset1}]\!] \ \boxed{\text{CS}} \Rightarrow \ \mathit{kcset} \colon \mathit{ktype}}{E \vdash [\![[\mathit{size}]\mathit{constset1}]\!] \ \boxed{\text{CS}} \Rightarrow \ \textbf{constsets}([\mathit{kcset}^{\mathit{size}}]) \colon \textbf{types}([\mathit{ktype}^{\mathit{size}}])}$$

$$\boxed{\text{CS4}}\ \frac{E \vdash [\![\mathit{enumerated}]\!] \ \boxed{\text{EM}} \Rightarrow \ \textbf{enum}(\mathit{ktypeno}, \mathit{tagno})}{E \vdash [\![\mathit{enumerated}]\!] \ \boxed{\text{CS}} \Rightarrow \ \textbf{enum}(\mathit{ktypeno}, \mathit{tagno}) \colon \textbf{typeno}(\mathit{ktypeno})}$$

$$\boxed{\text{CS5}}\ \frac{E \vdash [\![\mathit{enumerated}]\!] \ \boxed{\text{EM}} \Rightarrow \ \textbf{string}(\mathit{ktypeno}, \mathit{tagnoseq})}{E \vdash [\![\mathit{enumerated}]\!] \ \boxed{\text{CS}} \Rightarrow \ \textbf{string}(\mathit{ktypeno}, \mathit{tagnoseq}) \colon \textbf{typeno}(\mathit{ktypeno})}$$

$$\boxed{\mathsf{CS6}}\ \frac{E \vdash [\mathit{constset2}] \Rrightarrow\!\boxed{\mathrm{CS}}\!\Rightarrow \mathit{kcset}\colon \mathit{ktype}_1 \qquad \mathit{Find\text{-}Assoc}\ (\mathit{altname}, E) = \mathit{ktype}_2 \qquad \mathit{ktype}_1 =\!\boxed{\mathrm{T}}\!\models \mathit{ktype}_2 \qquad \mathit{Find\text{-}Alt}\ (\mathit{altname}, E) = \mathbf{enum}(\mathit{ktypeno}, \mathit{tagno})}{E \vdash [\mathit{altname}\&\mathit{constset2}] =\!\boxed{\mathrm{CS}}\!\Rightarrow \mathbf{constsetassoc}(\ \mathbf{enum}(\mathit{ktypeno}, \mathit{tagno})\ , \mathit{kcset})\colon\ \mathbf{typeno}(\mathit{ktypeno})}$$

$$\boxed{\mathsf{CS7}}\ \frac{E \vdash [\mathit{constset}_i] =\!\boxed{\mathrm{CS}}\!\Rightarrow \mathit{kcset}_i\colon \mathit{ktype}_i \qquad \mathit{Forall}\ i \in \{1..k\}}{E \vdash [(\mathit{constset}_1, \cdots, \mathit{constset}_k)] =\!\boxed{\mathrm{CS}}\!\Rightarrow \mathit{ConstsetTuple}([\mathit{kcset}_1, \cdots, \mathit{kcset}_k])\colon \mathit{TypeTuple}([\mathit{ktype}_1, \cdots, \mathit{ktype}_k])}$$

$$\boxed{\mathsf{CS8}}\ \frac{E \vdash [\mathit{type}] =\!\boxed{\mathrm{T}}\!\Rightarrow \mathit{ktype}}{E \vdash [\mathit{type}] =\!\boxed{\mathrm{CS}}\!\Rightarrow \ \mathbf{constsetany}(\mathit{ktype})\colon \mathit{ktype}}$$

3.5.5 Units

The complete Core ELLA unit syntax is given by

unit	::=	unit CONC unit1
	|	unit1
unit1	::=	STRING [size] unit1
	|	[size] unit1
	|	fnname unit1
	|	altname & unit1
	|	unit2 // altname
	|	unit2
unit2	::=	signalname
	|	enumerated
	|	unit2 [index]
	|	unit2 [index_{lwb} .. index_{upb}]
	|	unit2 [[unit]]
	|	REPLACE (unit, unit, unit)
	|	? type
	|	closedclause

with the transformations defined by

$$\boxed{\mathsf{U1}}\ \frac{\mathit{Find\text{-}Signal}\ (\mathit{signalname}, E) = \ \mathbf{signal}(\mathit{signalno}),\ \mathit{ktype}}{E \vdash [\mathit{signalname}] =\!\boxed{\mathrm{U}}\!\Rightarrow \ \mathbf{signal}(\mathit{signalno})\colon\ \mathit{ktype},\ E}$$

$$\boxed{\mathsf{U2}}\ \frac{E \vdash [\mathit{enumerated}] =\!\boxed{\mathrm{EM}}\!\Rightarrow \ \mathbf{enum}(\mathit{ktypeno}, \mathit{tagno})}{E \vdash [\mathit{enumerated}] =\!\boxed{\mathrm{U}}\!\Rightarrow \ \mathbf{enum}(\mathit{ktypeno}, \mathit{tagno})\colon\ \mathbf{typeno}(\mathit{ktypeno}),\ E}$$

$$\boxed{\text{U3}}\ \frac{E \vdash [\![\mathit{enumerated}]\!] \dashv\!\boxed{\text{EM}}\!\Rightarrow\ \mathbf{string}(\mathit{ktypeno}, \mathit{tagnoseq})}{E \vdash [\![\mathit{enumerated}]\!] \dashv\!\boxed{\text{U}}\!\Rightarrow\ \mathbf{string}(\mathit{ktypeno}, \mathit{tagnoseq}):\ \mathbf{typeno}(\mathit{ktypeno}),\ E}$$

$$\boxed{\text{U4}}\ \frac{\begin{array}{c} E \vdash [\![\mathit{unit}]\!] \dashv\!\boxed{\text{U}}\!\Rightarrow \mathit{kunit}_1:\ \mathit{ktype}_1,\ E_1 \\ E_1 \vdash [\![\mathit{unit1}]\!] \dashv\!\boxed{\text{U}}\!\Rightarrow \mathit{kunit}_2:\ \mathit{ktype}_2,\ E_2 \\ \mathit{ktype}_{out} = \mathit{Conc}(\mathit{ktype}_1,\ \mathit{ktype}_2) \end{array}}{E \vdash [\![\mathit{unit\ CONC\ unit1}]\!] \dashv\!\boxed{\text{U}}\!\Rightarrow\ \mathbf{conc}(\mathit{kunit}_1, \mathit{kunit}_2, \mathit{ktype}_{out}):\ \mathit{ktype}_{out},\ E_2}$$

$$\boxed{\text{U5}}\ \frac{\begin{array}{c} E \vdash [\![\mathit{unit1}]\!] \dashv\!\boxed{\text{U}}\!\Rightarrow \mathit{kunit}:\ \mathbf{typeno}(\mathit{ktypeno}),\ E' \\ \mathit{Is\text{-}Char}\ (\mathit{ktypeno},\ E) \end{array}}{\begin{array}{c} E \vdash [\![\mathit{STRING}[\mathit{size}]\mathit{unit1}]\!] \dashv\!\boxed{\text{U}}\!\Rightarrow \\ \mathbf{unitstring}(\mathit{size}, \mathit{kunit}):\ \mathbf{stringtype}(\mathit{size},\ \mathbf{typeno}(\mathit{ktypeno})),\ E' \end{array}}$$

$$\boxed{\text{U6}}\ \frac{E \vdash [\![\mathit{unit1}]\!] \dashv\!\boxed{\text{U}}\!\Rightarrow \mathit{kunit}:\ \mathit{ktype},\ E'}{E \vdash [\![[\mathit{size}]\mathit{unit1}]\!] \dashv\!\boxed{\text{U}}\!\Rightarrow\ \mathbf{units}([\mathit{kunit}^{\mathit{size}}]):\ \mathbf{types}([\mathit{ktype}^{\mathit{size}}]),\ E'}$$

$$\boxed{\text{U7}}\ \frac{\begin{array}{c} E \vdash [\![\mathit{unit1}]\!] \dashv\!\boxed{\text{U}}\!\Rightarrow \mathit{kunit}:\ \mathit{ktype},\ E' \\ \mathit{fnno} = \mathit{Find\text{-}Fn}\ (\mathit{fnname}, E) \\ (E.\mathit{fndec})[\mathit{fnno}].\mathit{inputtype} \dashv\!\boxed{\text{T}}\!\vDash \mathit{ktype} \end{array}}{E \vdash [\![\mathit{fnname\ unit1}]\!] \dashv\!\boxed{\text{U}}\!\Rightarrow\ \mathbf{instance}(\mathit{fnno}, \mathit{kunit}):\ ((E.\mathit{fndec})[\mathit{fnno}].\mathit{outputtype}),\ E'}$$

$$\boxed{\text{U8}}\ \frac{\begin{array}{c} E \vdash [\![\mathit{unit1}]\!] \dashv\!\boxed{\text{U}}\!\Rightarrow \mathit{kunit}_1:\ \mathit{ktype}_1,\ E' \\ \mathit{Find\text{-}Assoc}\ (\mathit{altname}, E)\ = \mathit{ktype}_2 \\ \mathit{ktype}_1 \dashv\!\boxed{\text{T}}\!\vDash \mathit{ktype}_2 \qquad \mathit{Find\text{-}Alt}\ (\mathit{altname}, E)\ =\ \mathbf{enum}(\mathit{ktypeno}, \mathit{tagno}) \end{array}}{\begin{array}{c} E \vdash [\![\mathit{altname}\&\mathit{unit1}]\!] \dashv\!\boxed{\text{U}}\!\Rightarrow \\ \mathbf{unitassoc}(\ \mathbf{enum}(\mathit{ktypeno}, \mathit{tagno}),\ \mathit{kunit}_1):\ \ \mathbf{typeno}(\mathit{ktypeno}),\ E' \end{array}}$$

$$\boxed{\text{U9}}\ \frac{\begin{array}{c} E \vdash [\![\mathit{unit2}]\!] \dashv\!\boxed{\text{U}}\!\Rightarrow \mathit{kunit}:\ \mathit{ktype},\ E' \qquad \mathit{Find\text{-}Assoc}\ (\mathit{altname}, E)\ = \mathit{ktype}, \\ \mathit{Find\text{-}Alt}\ (\mathit{altname}, E)\ =\ \mathbf{enum}(\mathit{typeno}, \mathit{index}), \\ \mathbf{typeno}(\mathit{typeno}) \dashv\!\boxed{\text{T}}\!\vDash \mathit{ktype} \end{array}}{E \vdash [\![\mathit{unit2//altname}]\!] \dashv\!\boxed{\text{U}}\!\Rightarrow\ \mathbf{extract}(\mathit{kunit},\ \mathbf{enum}(\mathit{typeno}, \mathit{index})):\ \mathit{ktype},\ E'}$$

$$\boxed{\text{U10}}\ \frac{\begin{array}{c} E \vdash [\![\mathit{unit2}]\!] \dashv\!\boxed{\text{U}}\!\Rightarrow \mathit{kunit}:\ \mathit{ktype},\ E' \qquad \mathit{Index}(\mathit{ktype}, \mathit{index}) = t \\ \mathit{Find\text{-}Integer\text{-}Type}(t, E') = \mathit{ktypeno},\ l,\ u \qquad l \leq \mathit{index} \leq u \end{array}}{E \vdash [\![\mathit{unit2}[\mathit{index}]]\!] \dashv\!\boxed{\text{U}}\!\Rightarrow\ \mathbf{index}(\mathit{kunit}, \mathit{index}, t):\ t,\ E'}$$

$$\boxed{\text{U11}}\ \frac{\begin{array}{c} E \vdash [\![\mathit{unit2}]\!] \dashv\!\boxed{\text{U}}\!\Rightarrow \mathit{kunit}:\ \mathit{ktype},\ E' \\ \mathit{Index}(\mathit{ktype}, \mathit{index}_{lwb})\ =\ t \qquad \mathit{Index}(\mathit{ktype}, \mathit{index}_{upb})\ =\ t \\ \mathit{Find\text{-}Integer\text{-}Type}(t, E')\ =\ \mathit{ktypeno},\ l,\ u \qquad l \leq \mathit{index}_{lwb} \leq \mathit{index}_{upb} \leq u \end{array}}{E \vdash [\![\mathit{unit2}[\mathit{index}_{lwb}..\mathit{index}_{upb}]]\!] \dashv\!\boxed{\text{U}}\!\Rightarrow\ \mathbf{trim}(\mathit{kunit}, \mathit{index}_{lwb}, \mathit{index}_{upb}, t):\ t,\ E'}$$

$$\boxed{\mathrm{U12}}\ \frac{\begin{array}{c} E \vdash [\![unit2]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow kunit_1\text{: } ktype_1,\ E_1 \\ E_1 \vdash [\![unit]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow kunit_2\text{: } ktype_2,\ E' \\ \textit{Find-Integer-Type}(ktype_2, E') = ktypeno,\ l,\ u \\ \textit{Find-Row}(ktype_1) = size,\ tt \qquad 1 \leq l \leq u \leq size \end{array}}{E \vdash [\![unit2[[unit]]]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow\ \mathbf{dyindex}(kunit_1, kunit_2, tt)\text{: } tt,\ E'}$$

$$\boxed{\mathrm{U13}}\ \frac{\begin{array}{c} E \vdash [\![unit_1]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow kunit_1\text{: } ktype_1,\ E_1 \\ E_1 \vdash [\![unit_2]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow kunit_2\text{: } ktype_2,\ E_2 \\ E_2 \vdash [\![unit_3]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow kunit_3\text{: } ktype_3,\ E' \\ \textit{Find-Integer-Type}(ktype_2,\ E_2) = ktypeno,\ l,\ u \\ \textit{Find-Row}(ktype_1) = size,\ t \\ 1 \leq l \leq u \leq size \qquad ktype_3 =\!\boxed{\mathrm{T}}\!= t \end{array}}{E \vdash [\![REPLACE(unit_1, unit_2, unit_3)]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow\ \mathbf{replace}(kunit_1, kunit_2, kunit_3)\text{: } ktype_1,\ E'}$$

$$\boxed{\mathrm{U14}}\ \frac{E \vdash [\![type]\!] =\!\boxed{\mathrm{T}}\!\Rightarrow ktype}{E \vdash [\![?type]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow\ \mathbf{unitquery}(ktype)\text{: } ktype,\ E}$$

$$\boxed{\mathrm{U15}}\ \frac{E \vdash [\![closedclause]\!] =\!\boxed{\mathrm{CC}}\!\Rightarrow kunit\text{: } ktype,\ E'}{E \vdash [\![closedclause]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow kunit\text{: } ktype,\ E'}$$

3.5.6 Closedclause

Closed clauses are given by

closedclause	::=	CASE unit OF cases ELSE unit ESAC
	|	($\text{unit}_1, \cdots, \text{unit}_k$)
	|	BEGIN $\text{step}_1 \cdots \text{step}_{k\text{-}1}$ OUTPUT unit END
	|	()
cases	::=	constset_1 : $\text{unit}_1, \cdots,$ constset_k : unit_k
step	::=	typedec
	|	fndec
	|	LET signalname = unit .
	|	MAKE fnname : signalname .
	|	JOIN unit → signalname .

with the transformations given by

$$\boxed{\mathrm{CA}}\ \frac{\begin{array}{c} E \vdash [\![constset]\!] =\!\boxed{\mathrm{CS}}\!\Rightarrow kconstset\text{: } ktypeconst \\ E \vdash [\![unit]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow kunit\text{: } ktype,\ E' \end{array}}{E \vdash [\![constset\text{: } unit]\!] =\!\boxed{\mathrm{CA}}\!\Rightarrow kconstset,\ kunit\text{: } ktypeconst,\ ktype,\ E'}$$

$$\boxed{\text{CC1}}\ \dfrac{\begin{array}{c} E \vdash \llbracket unit_1 \rrbracket =\!\boxed{\text{U}}\!\Rightarrow kunit_{in}\colon ktype_{in},\ E_1 \\ E_i \vdash \llbracket case_i \rrbracket =\!\boxed{\text{CA}}\!\Rightarrow kcs_i,\ ku_i\colon ktypec_i,\ ktypeu_i,\ E_{i+1} \qquad ForAll\ i \in \{1..k\} \\ E_{k+1} \vdash \llbracket unit_2 \rrbracket =\!\boxed{\text{U}}\!\Rightarrow kunit_{out}\colon ktype_{out},\ E' \\ ktypec_i =\!\boxed{\text{T}}\!\!= ktype_{in} \qquad ktypeu_i =\!\boxed{\text{T}}\!\!= ktype_{out} \qquad ForAll\ i \in \{1..k\} \\ Disjoint(kcs_i,\ kcs_j) \qquad ForAll\ i,j \in \{1..k\}\ i \neq j \end{array}}{\begin{array}{c} E \vdash \llbracket CASE\ unit_1\ OF\ case_1, \cdots, case_k\ ELSE\ unit_2\ ESAC \rrbracket =\!\boxed{\text{CC}}\!\Rightarrow \\ \mathbf{caseclause}(kunit_{in}, [\ \mathbf{case}(kcs_1, ku_1), \cdots,\ \mathbf{case}(kcs_k, ku_k)], kunit_{out})\colon ktype_{out},\ E' \end{array}}$$

$$\boxed{\text{CC2}}\ \dfrac{E_i \vdash \llbracket unit_i \rrbracket =\!\boxed{\text{U}}\!\Rightarrow kunit_i\colon ktype_i,\ E_{i+1} \qquad ForAll\ i \in \{1..k\}}{\begin{array}{c} E_1 \vdash \llbracket (unit_1, \cdots, unit_k) \rrbracket =\!\boxed{\text{CC}}\!\Rightarrow \\ UnitTuple([kunit_1, \cdots, kunit_k])\colon\ TypeTuple([ktype_1, \cdots, ktype_k]),\ E_{k+1} \end{array}}$$

$$\boxed{\text{CC3}}\ \dfrac{\begin{array}{c} Scope\text{-}Begin\ (E)\ = E_1 \\ E_i \vdash \llbracket step_i \rrbracket =\!\boxed{\text{SP}}\!\Rightarrow E_{i+1} \qquad ForAll\ i \in \{1..k\text{-}1\} \\ E_k \vdash \llbracket unit \rrbracket =\!\boxed{\text{U}}\!\Rightarrow kunit\colon\ ktype,\ E_{k+1} \\ Not\text{-}Local\text{-}Type(ktype, E_{k+1}) \qquad Check\text{-}Joins\ (E_{k+1}) \qquad Scope\text{-}End\ (E, E_{k+1}) = E' \end{array}}{E \vdash \llbracket BEGIN\ step_1 \cdots step_{k\text{-}1}\ OUTPUT\ unit\ END \rrbracket =\!\boxed{\text{CC}}\!\Rightarrow kunit\colon\ ktype,\ E'}$$

$$\boxed{\text{CC4}}\ \dfrac{}{E \vdash \llbracket\ ()\ \rrbracket =\!\boxed{\text{CC}}\!\Rightarrow \mathbf{unitvoid}\colon\ \mathbf{typevoid},\ E}$$

$$\boxed{\text{SP1}}\ \dfrac{\begin{array}{c} Scope\text{-}Fn\text{-}Begin(E) = E_1 \\ E_1 \vdash \llbracket typedec \rrbracket =\!\boxed{\text{TD}}\!\Rightarrow E_2 \\ Scope\text{-}Fn\text{-}End(E, E_2) = E' \end{array}}{E \vdash \llbracket typedec \rrbracket =\!\boxed{\text{SP}}\!\Rightarrow E'}$$

$$\boxed{\text{SP2}}\ \dfrac{\begin{array}{c} Scope\text{-}Fn\text{-}Begin(E) = E_1 \\ E_1 \vdash \llbracket fndec \rrbracket =\!\boxed{\text{FD}}\!\Rightarrow E_2 \\ Scope\text{-}Fn\text{-}End(E, E_2) = E' \end{array}}{E \vdash \llbracket fndec \rrbracket =\!\boxed{\text{SP}}\!\Rightarrow E'}$$

$$\boxed{\text{SP3}}\ \dfrac{E \vdash \llbracket unit \rrbracket =\!\boxed{\text{U}}\!\Rightarrow kunit\colon\ ktype,\ E'}{\begin{array}{c} E \vdash \llbracket LET\ signalname = unit. \rrbracket =\!\boxed{\text{SP}}\!\Rightarrow \\ Add\text{-}Signal\ (\ \mathbf{signaldec}(signalname, ktype, kunit),\ \mathbf{joined}, E') \end{array}}$$

$$\boxed{\text{SP4}}\ \dfrac{\begin{array}{c} Find\text{-}Fn\ (fnname, E)\ = fnno \\ ktype_{in} = (E.fndec)[fnno].inputtype \\ ktype_{out} = (E.fndec)[fnno].outputtype \end{array}}{\begin{array}{c} E \vdash \llbracket MAKE\ fnname\colon signalname. \rrbracket =\!\boxed{\text{SP}}\!\Rightarrow \\ Add\text{-}Signal\ (\ \mathbf{signaldec}(signalname, ktype_{out},\ \mathbf{instance}(fnno,\ \mathbf{unitquery}(ktype_{in}))),\ \mathbf{unjoined}, E) \end{array}}$$

$$\boxed{\mathsf{SP5}}\ \frac{\begin{array}{c} E \vdash [\![\mathit{unit}]\!] \Rightarrow\!\boxed{\mathrm{U}}\!\Rightarrow \mathit{kunit}\colon \mathit{ktype},\ E' \\ \textit{Find-Unjoined}\ (\mathit{signalname}, E)\ = \mathit{fnno} \\ \textit{Find-Signal}\ (\mathit{signalname}, E)\ =\ \mathbf{signal}(\mathit{signalno})\colon \mathit{ktype}_{out} \\ (E.\mathit{fndec})[\mathit{fnno}].\mathit{inputtype} =\!\boxed{\mathrm{T}}\!= \mathit{ktype} \end{array}}{\begin{array}{c} E \vdash [\![\mathit{JOIN\ unit} \longrightarrow \mathit{signalname.}]\!] =\!\boxed{\mathrm{SP}}\!\Rightarrow \\ \textit{Add-Join}\ (\ \mathbf{signaldec}(\mathit{signalname}, \mathit{ktype}_{out},\ \mathbf{instance}(\mathit{fnno}, \mathit{kunit})), \mathit{signalno}, E') \end{array}}$$

3.5.7 Built-In Functions

Built-in-functions (function bodies) are defined to be

functionbody ::= unit
| REFORM
| BIOP biopname
| DELAY (initialvalue, ambigtime, ambigvalue, delaytime)
| IDELAY (initialvalue, delaytime)
| SAMPLE (interval, initialvalue, skewtime)
| RAM (initialvalue)

with the following transformations

$$\boxed{\mathsf{BI1}}\ \frac{\mathit{TypeTuple}(\mathit{Flatten}(\mathit{ktype}_{in})) =\!\boxed{\mathrm{T}}\!= \mathit{TypeTuple}(\mathit{Flatten}(\mathit{ktype}_{out}))}{E \vdash [\![\mathit{REFORM}]\!]\ \{\mathit{ktype}_{in}, \mathit{ktype}_{out}\} =\!\boxed{\mathrm{BI}}\!\Rightarrow\ \mathbf{reform}}$$

$$\boxed{\mathsf{BI2}}\ \frac{\textit{Find-Biop}(\mathit{biopname},\ \mathit{ktype}_{in},\ \mathit{ktype}_{out}) =\ \mathbf{biop}(\mathit{biopname})}{E \vdash [\![\mathit{BIOP\ biopname}]\!]\ \{\mathit{ktype}_{in}, \mathit{ktype}_{out}\} =\!\boxed{\mathrm{BI}}\!\Rightarrow\ \mathbf{biop}(\mathit{biopname})}$$

$$\boxed{\mathsf{BI3}}\ \frac{\begin{array}{c} E \vdash [\![\mathit{initialvalue}]\!] =\!\boxed{\mathrm{C}}\!\Rightarrow \mathit{kconst}_i\colon \mathit{ktype}_i \\ E \vdash [\![\mathit{ambigvalue}]\!] =\!\boxed{\mathrm{C}}\!\Rightarrow \mathit{kconst}_a\colon \mathit{ktype}_a \\ \mathit{ktype}_{in} =\!\boxed{\mathrm{T}}\!= \mathit{ktype}_{out} =\!\boxed{\mathrm{T}}\!= \mathit{ktype}_i =\!\boxed{\mathrm{T}}\!= \mathit{ktype}_a \\ \mathit{ambigtime} \leq \mathit{delaytime} \end{array}}{\begin{array}{c} E \vdash [\![\mathit{DELAY}(\mathit{initialvalue}, \mathit{ambigtime}, \mathit{ambigvalue}, \mathit{delaytime})]\!]\ \{\mathit{ktype}_{in}, \mathit{ktype}_{out}\} =\!\boxed{\mathrm{BI}}\!\Rightarrow \\ \mathbf{delay}(\mathit{kconst}_i, \mathit{ambigtime}, \mathit{kconst}_a, \mathit{delaytime}) \end{array}}$$

$$\boxed{\mathsf{BI4}}\ \frac{E \vdash [\![\mathit{initialvalue}]\!] =\!\boxed{\mathrm{C}}\!\Rightarrow \mathit{kconst}\colon \mathit{ktype} \qquad \mathit{ktype} =\!\boxed{\mathrm{T}}\!= \mathit{ktype}_{in} =\!\boxed{\mathrm{T}}\!= \mathit{ktype}_{out}}{\begin{array}{c} E \vdash [\![\mathit{IDELAY}(\mathit{initialvalue}, \mathit{delaytime})]\!]\ \{\mathit{ktype}_{in}, \mathit{ktype}_{out}\} =\!\boxed{\mathrm{BI}}\!\Rightarrow \\ \mathbf{idelay}(\mathit{kconst}, \mathit{delaytime}) \end{array}}$$

$$\boxed{\mathsf{BI5}}\ \frac{\begin{array}{c} E \vdash [\![\mathit{initialvalue}]\!] =\!\boxed{\mathrm{C}}\!\Rightarrow \mathit{kconst}\colon \mathit{ktype} \\ \text{-}\mathit{interval} \leq \mathit{skew} \leq \mathit{interval} \qquad \mathit{ktype}_{in} =\!\boxed{\mathrm{T}}\!= \mathit{ktype}_{out} =\!\boxed{\mathrm{T}}\!= \mathit{ktype} \end{array}}{\begin{array}{c} E \vdash [\![\mathit{SAMPLE}(\mathit{interval}, \mathit{initialvalue}, \mathit{skew})]\!]\ \{\mathit{ktype}_{in}, \mathit{ktype}_{out}\} =\!\boxed{\mathrm{BI}}\!\Rightarrow \\ \mathbf{sample}(\mathit{interval}, \mathit{kconst}, \mathit{skew}) \end{array}}$$

$$\frac{\begin{array}{c} E \vdash [\![initialvalue]\!] \;\boxed{C}\!\Rightarrow\; kconst_I : ktype_I \\ ktype_{in} = \textbf{types}([ktype_{data}, ktype_{writeaddress}, ktype_{readaddress}, ktype_{writeenable}]) \\ ktype_{data} \;\boxed{T}\!= ktype_{out} \;\boxed{T}\!= ktype_I \\ \textit{Find-Integer-Type}\ (ktype_{writeaddress}) = -,\ lb,\ ub \\ \textit{Find-Integer-Type}\ (ktype_{readaddress}) = -,\ lb,\ ub \qquad lb = 1 \\ \textit{Check-Two-Val}\ (\textit{Get-Type} ktype_{writeenable}) \end{array}}{E \vdash [\![RAM(initialvalue)]\!]\ \{ktype_{in}, ktype_{out}\} \;\boxed{BI}\!\Rightarrow\; \textbf{ram}(kconst_I)}\ \boxed{\text{BI6}}$$

3.5.8 Type Declarations

Type declarations are defined as

typedec	::=	TYPE typename = typeornew.
typeornew	::=	type
	|	new
new	::=	NEW tagname / (lwb .. upb)
	|	NEW (typealt_1 | ⋯ | typealt_k)
	|	NEW tagname ($'\text{char}_1$ | ⋯ | $'\text{char}_k$)
typealt	::=	altname & type
	|	altname

with transformations on them by

$$\frac{E \vdash [\![new]\!] \;\boxed{NW}\!\Rightarrow\; knew,\ E'}{E \vdash [\![TYPE\ typename = new.\ ]\!] \;\boxed{TD}\!\Rightarrow\; \textit{Add-Type}\ (\ \textbf{typedec}(typename, knew),\ E')}\ \boxed{\text{TD1}}$$

$$\frac{E \vdash [\![type]\!] \;\boxed{T}\!\Rightarrow\; ktype}{E \vdash [\![TYPE\ typename = type.\ ]\!] \;\boxed{TD}\!\Rightarrow\; \textit{Add-Type-Name}\ (typename, ktype, E)}\ \boxed{\text{TD2}}$$

$$\frac{lwb \leq upb}{\begin{array}{c} E \vdash [\![NEW\ tagname/(lwb..upb)]\!] \;\boxed{NW}\!\Rightarrow \\ \textbf{ellaint}(tagname, lwb, upb),\ \textit{Add-Tag}\ (tagname, E) \end{array}}\ \boxed{\text{NW1}}$$

$$\frac{E_i \vdash [\![typealt_i]\!] \;\boxed{TA}\!\Rightarrow\; t_i,\ E_{i+1} \qquad \textit{ForAll}\ i \in \{1..k\}}{E_1 \vdash [\![NEW\ (typealt_1 \mid \cdots \mid typealt_k)]\!] \;\boxed{NW}\!\Rightarrow\; \textbf{tags}([t_1, \cdots, t_k]),\ E_{k+1}}\ \boxed{\text{NW2}}$$

$$\frac{char_i \neq char_j \qquad \textit{ForAll}\ i,j \in \{1..k\}\ i \neq j}{\begin{array}{c} E \vdash [\![NEW\ tagname('char_1 \mid \cdots \mid 'char_k)]\!] \;\boxed{NW}\!\Rightarrow \\ \textbf{chars}(tagname, [char_1, \cdots, char_k]),\ \textit{Add-Tag}\ (tagname, E) \end{array}}\ \boxed{\text{NW3}}$$

$$\boxed{\mathsf{TA1}}\ \frac{E \vdash [\![\mathit{type}]\!] =\!\boxed{\mathrm{T}}\!\Rightarrow \mathit{ktype}}{E \vdash [\![\mathit{altname\&type}]\!] =\!\boxed{\mathrm{TA}}\!\Rightarrow \ \mathbf{tag}(\mathit{altname}, \mathit{ktype}),\ \mathit{Add\text{-}Tag}\ (\mathit{altname}, E)}$$

$$\boxed{\mathsf{TA2}}\ \frac{}{E \vdash [\![\mathit{altname}]\!] =\!\boxed{\mathrm{TA}}\!\Rightarrow \ \mathbf{tag}(\mathit{altname}, \{\mathsf{nil}\}),\ \mathit{Add\text{-}Tag}\ (\mathit{altname}, E)}$$

3.5.9 Function Declarations

Function declarations are given by

fndec	::=	FN fnname = input → type : functionbody.
input	::=	(type_1 : signalname_1, $\cdots$, type_k : signalname_k)
	|	()

and the transformations on them by

$$\boxed{\mathsf{FD1}}\ \frac{\begin{array}{c} E \vdash [\![\mathit{input}]\!] =\!\boxed{\mathrm{IN}}\!\Rightarrow \mathit{ktype}_{\mathit{inputs}},\ E' \\ E' \vdash [\![\mathit{type}]\!] =\!\boxed{\mathrm{T}}\!\Rightarrow \mathit{ktype}_{\mathit{out}} \\ E' \vdash [\![\mathit{unit}]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow \mathit{kunit}\colon \mathit{ktype},\ E'' \\ \mathit{ktype}_{\mathit{out}} =\!\boxed{\mathrm{T}}\!\models \mathit{ktype} \end{array}}{\begin{array}{c} E \vdash [\![\mathit{FN\ fnname} = \mathit{input} \longrightarrow \mathit{type}\colon \mathit{unit}.\ ]\!] =\!\boxed{\mathrm{FD}}\!\Rightarrow \\ \mathit{Add\text{-}Fn}\ (\ \mathbf{fndec}(\mathit{fnname}, \mathit{ktype}_{\mathit{inputs}},\ E''.\mathit{sigdec}, \mathit{ktype}_{\mathit{out}}, \mathit{kunit}),\ E,\ E'') \end{array}}$$

$$\boxed{\mathsf{FD2}}\ \frac{\begin{array}{c} E \vdash [\![\mathit{input}]\!] =\!\boxed{\mathrm{IN}}\!\Rightarrow \mathit{ktype}_{\mathit{inputs}},\ E' \\ E' \vdash [\![\mathit{type}]\!] =\!\boxed{\mathrm{T}}\!\Rightarrow \mathit{ktype}_{\mathit{out}} \\ E' \vdash [\![\mathit{builtin}]\!]\ \{\mathit{ktype}_{\mathit{inputs}}, \mathit{ktype}\} =\!\boxed{\mathrm{BI}}\!\Rightarrow \mathit{kbuiltin} \end{array}}{\begin{array}{c} E \vdash [\![\mathit{FN\ fnname} = \mathit{input} \longrightarrow \mathit{type}\colon \mathit{builtin}.\ ]\!] =\!\boxed{\mathrm{FD}}\!\Rightarrow \\ \mathit{Add\text{-}Fn}\ (\ \mathbf{fndec}(\mathit{fnname}, \mathit{ktype}_{\mathit{inputs}}, [\,], \mathit{ktype}_{\mathit{out}}, \mathit{kbuiltin}),\ E,\ E') \end{array}}$$

$$\boxed{\mathsf{IN1}}\ \frac{\begin{array}{c} E_1 \vdash [\![\mathit{type}_i]\!] =\!\boxed{\mathrm{T}}\!\Rightarrow \mathit{ktype}_i \qquad \mathit{ForAll}\ i \in \{1..k\} \\ \mathit{Add\text{-}Signal}(\ \mathbf{signaldec}(\mathit{signalname}_i, \mathit{ktype}_i,\ \mathbf{input}),\ \mathbf{joined}, E_i) = E_{i+1} \quad \mathit{ForAll}\ i \in \{1..k\} \end{array}}{\begin{array}{c} E_1 \vdash [\![(\mathit{type}_1\colon \mathit{signalname}_1, \cdots, \mathit{type}_k\colon \mathit{signalname}_k)]\!] =\!\boxed{\mathrm{IN}}\!\Rightarrow \\ \mathit{TypeTuple}([\mathit{ktype}_1, \cdots, \mathit{ktypes}_k]),\ E_{k+1} \end{array}}$$

$$\boxed{\mathsf{IN2}}\ \frac{}{E \vdash [\![\ ()\ ]\!] =\!\boxed{\mathrm{IN}}\!\Rightarrow \ \mathbf{typevoid},\ E}$$

3.5.10 Closure

A Closure is defined to be

declaration	::=	typedec
		| fndec
closure	::=	declaration$_1$ $\cdots$ declaration$_k$

with the following transforms

$$\boxed{\mathsf{D1}}\ \frac{E \vdash [\![typedec]\!] \ \Rightarrow\boxed{\mathrm{TD}}\Rightarrow\ E'}{E \vdash [\![typedec]\!] \ \Rightarrow\boxed{\mathrm{D}}\Rightarrow\ E'}$$

$$\boxed{\mathsf{D2}}\ \frac{E \vdash [\![fndec]\!] \ \Rightarrow\boxed{\mathrm{FD}}\Rightarrow\ E'}{E \vdash [\![fndec]\!] \ \Rightarrow\boxed{\mathrm{D}}\Rightarrow\ E'}$$

$$\boxed{\mathsf{CL}}\ \frac{E_i \vdash [\![declaration_i]\!] \ \Rightarrow\boxed{\mathrm{D}}\Rightarrow\ E_{i+1} \qquad ForAll\ i \in \{1..k\}}{E_1 \vdash [\![declaration_1 \cdots declaration_k]\!] \ \Rightarrow\boxed{\mathrm{CL}}\Rightarrow\ E_{k+1}}$$

$$\boxed{\mathsf{KERNEL}}\ \frac{InitialEnv \vdash [\![closure]\!] \ \Rightarrow\boxed{\mathrm{CL}}\Rightarrow\ E}{[\![closure]\!] \ \Rightarrow\boxed{\mathrm{KERNEL}}\Rightarrow\ (E.typedec, E.fndec)}$$

3.6 Extracting the Type of a Kernel Structure

This section gives an example of how the **Kernel** can be used to find information about a circuit description. Three functions are defined which extract the type information from constants, constant sets and unit expressions.

3.6.1 Constant Type Value

This section defines the function Type-Of-Const for getting the Type of a constant expression

Type-Of-Const : *kConst* $\rightarrow$ *kType*

Type-Of-Const(*const*) $\triangleq$

  cases *const* **of**

enum(*typeno*, _)	$\rightarrow$	**typeno**(*typeno*)
string(*typeno*, [$tag_1, \cdots, tag_n$])	$\rightarrow$	**stringtype**(n, **typeno**(*typeno*))
conststring(*size*, c)	$\rightarrow$	**stringtype**(*size*, *Type-Of-Const*(c))
consts([$c_1, \cdots, c_n$])	$\rightarrow$	**types**([*Type-Of-Const* (c_1), $\cdots$, *Type-Of-Const* (c_n)])
constassoc(**enum**(*typeno*, _), _)	$\rightarrow$	**typeno**(*typeno*)
constquery(*type*)	$\rightarrow$	*type*
constvoid	$\rightarrow$	**typevoid**

  end

3.6.2 Constant Set Type Value

This section defines the function Type-Of-Constset for getting the Type of a constant set expression

Type-Of-Constset : *kConstset* → *kType*

Type-Of-Constset(*constset*) $\triangleq$
  cases *constset* **of**
  enum(*typeno*, _) → **typeno**(*typeno*)
  string(*typeno*, [$tag_1, \cdots, tag_n$]) → **stringtype**(n, **typeno**(*typeno*))
  constsetalts([c, ⋯]) → *Type-Of-Constset* (c)
  constsetstring(*size*, c) → **stringtype**(*size*, *Type-Of-Constset*(c))
  constsets([$c_1, \cdots, c_n$]) → **types**([*Type-Of-Constset* (c_1), ⋯, *Type-Of-Constset* (c_n)])
  constsetassoc(**enum**(*typeno*, _), _) → **typeno**(*typeno*)
  constsetany(*type*) → *type*
  end

3.6.3 Unit Type Value

This section defines the function Type-Of-Unit for getting the Type of a unit expression. The fndec sequence passed in is the sequence of fndecs in the closure up to and including the declaration of the outermost function in which the unit expression occurs.

Type-Of-Unit : $kFndec^* \times kUnit \to kType$

Type-Of-Unit(*f*, *unit*) $\triangleq$
  cases *unit* **of**
  enum(*typeno*, _) → **typeno**(*typeno*)
  string(*typeno*, [$tag_1, \cdots, tag_n$]) → **stringtype**(n, **typeno**(*typeno*))
  conc(_, _, *type*) → *type*
  unitstring(*size*, *u*) → **stringtype**(*size*, *Type-Of-Unit* (*f*, *u*))
  units([$u_1, \cdots, u_n$]) → **types**([*Type-Of-Unit* (f, u_1), ⋯, *Type-Of-Unit* (f, u_n)])
  instance(*fnno*, _) → *f*[*fnno*].*outputtype*
  unitassoc(**enum**(*typeno*, _), _) → **typeno**(*typeno*)
  extract(*u*, _) → *Type-Of-Unit* (*f*, *u*)
  signal(*signalno*) → (*f*[len *f*].*signaldec*)[*signalno*].*type*
  index(_, _, *outputtype*) → *outputtype*
  trim(_, _, _, *outputtype*) → *outputtype*
  dyindex(_, _, *outputtype*) → *outputtype*
  replace(*u*, _, _) → *Type-Of-Unit* (*f*, *u*)
  unitquery(*type*) → *type*
  caseclause(_, _, *u*) → *Type-Of-Unit*(*f*, *u*)
  unitvoid → **typevoid**
  end

3.7 A Formal Transformation Example

In this section we give an example of the transformation rules applied to a simple Core ELLA description. Through this example the reader will see how the scopes and types are incorporated into the environment and how the information is used.

We consider the following Core ELLA description

```
TYPE bool = NEW (t | f).

FN NOR = (bool:in1, bool:in2) -> bool:
CASE (in1, in2) OF
  (f,t):f,
  (t,f):f,
  (t,t):f,
 ELSE t
ESAC.


FN A = (bool:in1, bool:in2) -> bool:
BEGIN
  LET ip = (in1,in2).
  FN B = (bool:ip1, bool:ip2) -> bool: NOR(ip1,ip2).
  MAKE B:b.
  JOIN ip -> b.
  OUTPUT b
END.
```

with the initial environment

$$E = \mathbf{env}([\,],[\,],[\,],\{\},\{\},\{\},\{\},\{\},\{\},\{\},\{\},\{\},\{\})$$

For brevity the constructor name **env** will be omitted from the declarations of environments throughout this section.

We start be applying the transformation rules necessary for the type declaration 'bool' and then proceed to add the functions 'NOR' followed by 'A'. Throughout this example expressions which are numbered on the right hand side are expressions which need to be satisfied, their evaluation being shown by expressions with numbers on the left hand side.

3.7.1 Type Declaration

$$[\mathrm{TD1}]: E \vdash [\![NEW(t \mid f).]\!] \,=\!\boxed{\mathrm{NW}}\!\Rightarrow knew,\ E_2 \qquad (1)$$
$$E \vdash [\![TYPE\ bool = NEW(t \mid f).]\!] \,=\!\boxed{\mathrm{TD}}\!\Rightarrow Add\text{-}Type(\ \mathbf{typedec}(bool, knew),\ E_2) \qquad (2)$$

$$(1)\quad : E \vdash [\![t]\!] \,=\!\boxed{\mathrm{TA}}\!\Rightarrow ty_1,\ E_1 \qquad (3)$$

$$E_1 \vdash [\![f]\!] \overset{\boxed{\text{TA}}}{\Longrightarrow} ty_2,\ E_2 \qquad (4)$$
$$E \vdash [\![NEW(t \mid f).]\!] \overset{\boxed{\text{NW}}}{\Longrightarrow} \textbf{tags}([ty_1,\ ty_2]), E_2 \qquad (5)$$

(3) $: E \vdash [\![t]\!] \overset{\boxed{\text{TA}}}{\Longrightarrow} \textbf{tag}(t, \{\}), E_1$

where
E_1 = Add-Tag (t, E)

i.e.
$E_1 = ([], [], [], \{\}, \{\}, \{\}, \{t \mapsto \textbf{consttag}(1)\}, \{\}, \{\}, \{\}, \{\}, \{\})$

Also

(4) $: E_1 \vdash [\![f]\!] \overset{\boxed{\text{TA}}}{\Longrightarrow} \textbf{tag}(f, \{\}), E_2$

where
E_2 = Add-Tag (f, E_1)

i.e.
$E_2 = ([], [], [], \{\}, \{\}, \{\}, \{f \mapsto \textbf{consttag}(1), t \mapsto \textbf{consttag}(1)\}, \{\}, \{\}, \{\}, \{\}, \{\})$

(5) $: E \vdash [\![NEW(t| f)]\!] \overset{\boxed{\text{NW}}}{\Longrightarrow} \textbf{tags}([\ \textbf{tag}(t, \{\}),\ \textbf{tag}(f, \{\})]), E_2$

which gives

(2) $: E \vdash [\![TYPE\ bool = NEW(t \mid f).\ ]\!] \overset{\boxed{\text{TD}}}{\Longrightarrow} E_3$

where
$E_3 = \textit{Add-Type}(\ \textbf{typedec}(bool,\ \textbf{tags}([\ \textbf{tag}(t, \{\}),\ \textbf{tag}(f, \{\})])), E_2)$

i.e.

$$\begin{aligned} E_3 = (\ & [\ \textbf{typedec}(bool,\ \textbf{tags}([\ \textbf{tag}(t, \{\}),\ \textbf{tag}(f, \{\})]))], \\ & [], \\ & [], \\ & \{\}, \{\}, \\ & \{\}, \{bool \mapsto \textbf{typeno}(1), f \mapsto \textbf{consttag}(1), t \mapsto \textbf{consttag}(1)\}, \\ & \{\}, \{\}, \\ & \{\}, \{\}, \{\} \\ ) & \end{aligned}$$

we are now in the position where we can apply the transformations to the function NOR:

3.7.2 Function Declaration

In order to apply rule [FD] to function NOR we need the following rule applications

$$
\begin{array}{lll}
[\mathrm{FD1}]: & E_3 \vdash [\![(\mathit{bool}\colon \mathit{in}1, \mathit{bool}\colon \mathit{in}2)]\!] \;\boxed{\mathrm{IN}}\!\Rightarrow \mathit{ktype}_i, E_5 & (1)\\
& E_5 \vdash [\![\mathit{bool}]\!] \;\boxed{\mathrm{T}}\!\Rightarrow \mathit{ktype}_o & (2)\\
& E_5 \vdash [\![\mathit{CASE} \,..\, \mathit{ESAC}.]\!] \;\boxed{\mathrm{U}}\!\Rightarrow \mathit{kunit}\colon \mathit{ktype}_u,\ E_6 & (3)\\
& \mathit{ktype}_o \;\boxed{\mathrm{T}}\!\models \mathit{ktype}_u & (4)\\
& E_3 \vdash [\![\mathit{FN}\ \mathit{NOR}\ ...]\!] \;\boxed{\mathrm{FD}}\!\Rightarrow E_6 \equiv & \\
& \qquad \mathit{Add\text{-}Fn}(\ \mathbf{fndec}(\mathit{NOR}, \mathit{ktype}_i, E_5.\mathit{sigdec}, \mathit{ktype}_o, \mathit{kunit}), E_3, E_5) & (5)
\end{array}
$$

Now

$$
\begin{array}{lll}
(1) & : E_3 \vdash [\![\mathit{bool}]\!] \;\boxed{\mathrm{T}}\!\Rightarrow \mathit{ktype} & (6)\\
& \mathit{Add\text{-}Signal}(\ \mathbf{signaldec}(\mathit{in}1, \mathit{ktype},\ \mathbf{input}),\ \mathbf{joined}, E_3) \ =\ E_4 & (7)\\
& \mathit{Add\text{-}Signal}(\ \mathbf{signaldec}(\mathit{in}2, \mathit{ktype},\ \mathbf{input}),\ \mathbf{joined}, E_4) \ =\ E_5 & (8)\\
& E_3 \vdash [\![(\mathit{bool}\colon \mathit{in}1, \mathit{bool}\colon \mathit{in}2)]\!] \;\boxed{\mathrm{IN}}\!\Rightarrow \mathit{TypeTuple}[\mathit{ktype}, \mathit{ktype}], E_5 & (9)
\end{array}
$$

$$(6) \quad : E_3 \vdash [\![\mathit{bool}]\!] \;\boxed{\mathrm{T}}\!\Rightarrow \mathit{Find\text{-}Type}(\mathit{bool}, E_3) = \ \mathbf{typeno}(1)$$

$$(7) \quad : \mathit{Add\text{-}Signal}(\ \mathbf{signaldec}(\mathit{in}1,\ \mathbf{typeno}(1),\ \mathbf{input}),\ \mathbf{joined}, E_3) = E_4$$

where

$$
\begin{array}{ll}
\mathrm{E}_4 & = (\ [\ \mathbf{typedec}(\mathrm{bool},\ \ \mathbf{tags}([\ \mathbf{tag}(\mathrm{t},\{\}),\ \ \mathbf{tag}(\mathrm{f},\{\})]))\ ],\\
& \quad [\,],\\
& \quad [\ \mathbf{signaldec}(\mathrm{in1},\ \ \mathbf{typeno}(1),\ \ \mathbf{input})\ ],\\
& \quad \{\}, \{\},\\
& \quad \{\}, \{\mathit{bool} \mapsto\ \mathbf{typeno}(1), f \mapsto\ \mathbf{consttag}(1), t \mapsto\ \mathbf{consttag}(1)\},\\
& \quad \{\}, \{\mathit{in}1 \mapsto\ \mathbf{sig}(1,\ \mathbf{joined})\}\\
& \quad \{\}, \{\}, \{\}\\
& \ )
\end{array}
$$

$$(8) \quad : \mathit{Add\text{-}Signal}(\ \mathbf{signaldec}(\mathit{in}2,\ \mathbf{typeno}(1),\ \mathbf{input}),\ \mathbf{joined}, E_4) = E_5$$

where

$$
\begin{array}{ll}
\mathrm{E}_5 & = (\ [\ \mathbf{typedec}(\mathrm{bool},\ \ \mathbf{tags}([\ \mathbf{tag}(\mathrm{t},\{\}),\ \ \mathbf{tag}(\mathrm{f},\{\})]))\ ],\\
& \quad [\,],\\
& \quad [\ \mathbf{signaldec}(\mathrm{in1},\ \ \mathbf{typeno}(1),\ \ \mathbf{input}),\ \mathbf{signaldec}(\mathrm{in2},\ \ \mathbf{typeno}(1),\ \ \mathbf{input})\ ],\\
& \quad \{\}, \{\},\\
& \quad \{\}, \{\mathrm{bool} \mapsto\ \mathbf{typeno}(1), \mathrm{f} \mapsto\ \mathbf{consttag}(1), \mathrm{t} \mapsto\ \mathbf{consttag}(1)\},\\
& \quad \{\}, \{\mathrm{in2} \mapsto\ \mathbf{sig}(2,\ \mathbf{joined}), \mathrm{in1} \mapsto\ \mathbf{sig}(1,\ \mathbf{joined})\},\\
& \quad \{\}, \{\}, \{\}\\
& \ )
\end{array}
$$

the premises of IN are now satisfied, thus

(9) $: E_3 \vdash [\![(bool\colon in1, bool\colon in2)]\!] =\!\boxed{\mathrm{IN}}\!\Rightarrow$ $\mathbf{types}([\ \mathbf{typeno}(1),\ \mathbf{typeno}(1)]),\ E_5$

(2) $: E_5 \vdash [\![bool]\!] =\!\boxed{\mathrm{T}}\!\Rightarrow$ $\mathbf{typeno}(1)$

The evaluation of the CASE clause can now proceed :-

(3) $: E_5 \vdash [\![(in1, in2)]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow kunit_{in}\colon\ ktype_{in},\ E_5$ (10)
$E_5 \vdash [\![(f,t)\colon f]\!] =\!\boxed{\mathrm{CA}}\!\Rightarrow kcs_1,\ ku_1\colon\ ktypec_1,\ ktypeu_1,\ E_5$ (11)
$E_5 \vdash [\![(t,f)\colon f]\!] =\!\boxed{\mathrm{CA}}\!\Rightarrow kcs_2,\ ku_2\colon\ ktypec_2,\ ktypeu_2,\ E_5$ (12)
$E_5 \vdash [\![(t,t)\colon f]\!] =\!\boxed{\mathrm{CA}}\!\Rightarrow kcs_3,\ ku_3\colon\ ktypec_3,\ ktypeu_3,\ E_5$ (13)
$E_5 \vdash [\![t]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow kunit_o\colon\ ktype_o,\ E_5$ (14)
$ktypec_1\ =\!\boxed{\mathrm{T}}\!=\ ktypec_2\ =\!\boxed{\mathrm{T}}\!=\ ktypec_3$ (15)
$ktypeu_1\ =\!\boxed{\mathrm{T}}\!=\ ktypeu_2\ =\!\boxed{\mathrm{T}}\!=\ ktypeu_3\ =\!\boxed{\mathrm{T}}\!=\ ktype_o$ (16)
$Disjoint(kcs_1, kcs_2) \wedge Disjoint(kcs_2, kcs_3) \wedge Disjoint(kcs_1, kcs_3)$ (17)
$E_3 \vdash [\![CASE\ ..\ ESAC]\!] =\!\boxed{\mathrm{CC}}\!\Rightarrow$
$\mathbf{caseclause}(kunit_{in}, [\ \mathbf{case}(kcs_1, ku_1),\ \mathbf{case}(kcs_2, ku_2),$
$\mathbf{case}(kcs_3, ku_3)],\ kunit_o)\colon\ ktype_o,\ E_5$ (18)

Now

(10) $: E_5 \vdash [\![in1]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow kunit_1\colon\ ktype_1,\ E_5$ (19)
$E_5 \vdash [\![in2]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow kunit_2\colon\ ktype_2,\ E_5$ (20)
$E_5 \vdash [\![(in1, in2)]\!] =\!\boxed{\mathrm{CC}}\!\Rightarrow UnitTuple[kunit_1, kunit_2]\colon\ TypeTuple[ktype_1, ktype_2],\ E_5$ (21)
$E_5 \vdash [\![(in1, in2)]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow UnitTuple[kunit_1, kunit_2]\colon\ TypeTuple[ktype_1, ktype_2],\ E_5$ (22)

(19) $: E_5 \vdash [\![in1]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow$ $\mathbf{signal}(1)\colon\ \mathbf{typeno}(1),\ E_5$

(20) $: E_5 \vdash [\![in2]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow$ $\mathbf{signal}(2)\colon\ \mathbf{typeno}(1),\ E_5$

thus

(21) $: E_5 \vdash [\![(in1, in2)]\!] =\!\boxed{\mathrm{CC}}\!\Rightarrow$
$\mathbf{units}([\ \mathbf{signal}(1),\ \mathbf{signal}(2)])\colon\ \mathbf{types}([\ \mathbf{typeno}(1),\ \mathbf{typeno}(1)]),\ E_5$

(22) $: E_5 \vdash [\![(in1, in2)]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow$
$\mathbf{units}([\ \mathbf{signal}(1),\ \mathbf{signal}(2)])\colon\ \mathbf{types}([\ \mathbf{typeno}(1),\ \mathbf{typeno}(1)]),\ E_5$

For evaluation of case arm alternatives we need

(11) $: E_5 \vdash [\![(f,t)]\!] =\!\boxed{\mathrm{CS}}\!\Rightarrow kcs_1\colon\ ktypec_1$ (23)
$E_5 \vdash [\![f]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow ku_1\colon\ ktypeu_1,\ E_5$ (24)
$E_5 \vdash [\![(f,t)\colon f]\!] =\!\boxed{\mathrm{CA}}\!\Rightarrow kcs_1,\ ku_1\colon\ ktypec_1,\ ktypeu_1,\ E_5$ (25)

(23) $: E_5 \vdash [\![f]\!] =\!\boxed{\mathrm{CS}}\!\Rightarrow kc_1\colon\ kt_1$ (26)
$E_5 \vdash [\![t]\!] =\!\boxed{\mathrm{CS}}\!\Rightarrow kc_2\colon\ kt_2$ (27)
$E_5 \vdash [\![(f,t)]\!] =\!\boxed{\mathrm{CS}}\!\Rightarrow ConstsetTuple([kc_1, kc_2])\colon\ TypeTuple([kt_1, kt_2])$ (28)

(26) $: E_5 \vdash [\![f]\!] \dashv\!\boxed{\mathrm{CS}}\!\Rightarrow$ **enum**(1, 2), **typeno**(1)

(27) $: E_5 \vdash [\![t]\!] \dashv\!\boxed{\mathrm{CS}}\!\Rightarrow$ **enum**(1, 1), **typeno**(1)

combining these last two results

(28) $: E_5 \vdash [\![(f, t)]\!] \dashv\!\boxed{\mathrm{CS}}\!\Rightarrow$ **constsets**([**enum**(1, 2), **enum**(1, 1)]): **types**([**typeno**(1), **typeno**(1)]

(24) $: E_5 \vdash [\![f]\!] \dashv\!\boxed{\mathrm{U}}\!\Rightarrow$ **enum**(1, 2): **typeno**(1), E_5

thus

(25) $: E_5 \vdash [\![(f, t)\colon f)]\!] \dashv\!\boxed{\mathrm{CA}}\!\Rightarrow$ **constsets**([**enum**(1, 2), **enum**(1, 1)]), **enum**(1, 2):
types([**typeno**(1), **typeno**(1)]), **typeno**(1), E_5

Similarly

(12) $: E_5 \vdash [\![(t, f)\colon f)]\!] \dashv\!\boxed{\mathrm{CA}}\!\Rightarrow$ **constsets**([**enum**(1, 1), **enum**(1, 2)]), **enum**(1, 2):
types([**typeno**(1), **typeno**(1)]), **typeno**(1), E_5

(13) $: E_5 \vdash [\![(t, t)\colon f)]\!] \dashv\!\boxed{\mathrm{CA}}\!\Rightarrow$ **constsets**([**enum**(1, 1), **enum**(1, 1)]), **enum**(1, 2):
types([**typeno**(1), **typeno**(1)]), **typeno**(1), E_5

the ELSE clause of the CASE statement gives

(14) $: E_5 \vdash [\![t]\!] \dashv\!\boxed{\mathrm{U}}\!\Rightarrow$ **enum**(1, 1): **typeno**(1), E_5

(15), (16) and (17) are obviously true and hence we have

(18) $: E_5 \vdash [\![CASE....ESAC]\!] \dashv\!\boxed{\mathrm{CC}}\!\Rightarrow$
caseclause(**units**([**signal**(1), **signal**(2)]),
[**case**(**constsets**([**enum**(1, 2), **enum**(1, 1)]), **enum**(1, 2)),
case(**constsets**([**enum**(1, 1), **enum**(1, 2)]), **enum**(1, 2)),
case(**constsets**([**enum**(1, 1), **enum**(1, 1)]), **enum**(1, 2))],
enum(1, 1)): **typeno**(1), E_5

(4) : **typeno**(1) $\dashv\!\boxed{\mathrm{T}}\!\models$ **typeno**(1)

Bringing the above results together for the complete function gives:

(5) $: E_3 \vdash [\![FN\ NOR = ...]\!] \dashv\!\boxed{\mathrm{FD}}\!\Rightarrow E_6$

where

```
E6  = Add-Fn ( fndec  (NOR, types([ typeno(1), typeno(1)]),
                      [ signaldec(in1, typeno(1), input),
                          signaldec(in2, typeno(1), input)],
                       typeno(1),
                       caseclause( units([ signal(1), signal(2)]),
                         [ case( constsets([ enum(1,2), enum(1,1)]), enum(1,2)),
                           case( constsets([ enum(1,1), enum(1,2)]), enum(1,2)),
                           case( constsets([ enum(1,1), enum(1,1)]), enum(1,2))],
                           enum(1,1)),
                      E3, E5 )
```

i.e

```
E6  = ( [ typedec(bool, tags([ tag(t,{}), tag(f,{})]))],
        [ fndec(NOR,
               types([ typeno(1), typeno(1)]),
              [ signaldec(in1, typeno(1), input), signaldec(in2, typeno(1), input)],
               typeno(1),
               caseclause( units([ signal(1), signal(2)]),
                 [ case( constsets([ enum(1,2), enum(1,1)]), enum(1,2)),
                   case( constsets([ enum(1,1), enum(1,2)]), enum(1,2)),
                   case( constsets([ enum(1,1), enum(1,1)]), enum(1,2))],
                   enum(1,1))
        ]
        [],
        {}, {NOR ↦ 1},
        {}, {bool ↦ typeno(1), f ↦ consttag(1), t ↦ consttag(1)},
        {}, {},
        {}, {}, {}
      )
```

3.7.3 Function Declaration with Scoping

We now add to the above environment the function 'A' which will demonstrate the scoping mechanism of the transformation rules. For convenience we reproduce here the definition of function 'A'

```
FN A = (bool:in1, bool:in2) -> bool:
BEGIN
  LET ip = (in1,in2).
  FN B = (bool:ip1, bool:ip2) -> bool: NOR(ip1,ip2).
  MAKE B:b.
  JOIN ip -> b.
  OUTPUT b
END.
```

Assuming the result of the previous section, the starting environment for the transformation process of function 'A' is:

E = ([**typedec**(bool)], : Type declarations
[**fndec**(NOR)], : Fn declarations
[], : Signal declarations
{}, {NOR $\rightarrow$ 1}, : Fn mappings
{}, {bool $\rightarrow$, f$\rightarrow$, t$\rightarrow$ } : Type mappings
{}, {} : Signal mappings
{}, {}, {} : Out of scope Nms and Fns
)

In order to simplfy reading whenever ' **bool**' is used, i.e. in a non-syntactic position, it should be taken to be equal to ' **typeno**(1)'. Also the 'bool' type declaration and the NOR function declaration have been abbreviated to only the first field of the structures. The name-mapping field for the type declaration has also be abbreviated to only contain the domains.

The rule for adding function 'A' to the environment is given by [FD1] which requires the following to be satisfied

$$[\text{FD1}]: E \vdash [\![(bool\!: in1, bool\!: in2)]\!] =\!\boxed{\text{IN}}\!\Rightarrow ktype_i, E' \quad (1)$$

$$E' \vdash [\![bool]\!] =\!\boxed{\text{T}}\!\Rightarrow ktype_o \quad (2)$$

$$E' \vdash [\![BEGIN \;..\; END.]\!] =\!\boxed{\text{U}}\!\Rightarrow kunit\!: ktype_u,\; E'' \quad (3)$$

$$ktype_o =\!\boxed{\text{T}}\!= ktype_u \quad (4)$$

$$E \vdash [\![FN\ A\ ...]\!] =\!\boxed{\text{FD}}\!\Rightarrow E''' \equiv Add\text{-}Fn(\ \mathbf{fndec}(A, ktype_i, E''.sigdec, ktype_o, kunit), E, E'') \quad (5)$$

We now proceed to evaluate the different expressions in order to arrive at the final environment of E'''.

(1) : $ktype_i$ = **types**([**bool**, **bool**])

E' = ([**typedec**(bool)],
[**fndec**(NOR)],
[**signaldec**(in1, **bool**, **input**), **signaldec**(in2, **bool**, **input**)]
{}, {NOR $\rightarrow$ 1}
{}, {bool $\rightarrow$, f$\rightarrow$, t$\rightarrow$ }
{}, {in2$\rightarrow$ **sig**(2, **joined**), in1$\rightarrow$ **sig**(1, **joined**)}
{}, {}, {}
)

(2) : $ktype_o = Find\text{-}Type(bool, E') \equiv$ **bool** (= **typeno**(1))

(3) : $Scope\text{-}Begin(E') = E1$ (6)

$E1 \vdash [\![LET\ ip\ =\ (in1, in2).]\!] =\!\boxed{\text{SP}}\!\Rightarrow E2$ (7)

$E2 \vdash [\![FN\ B\ =\ ...]\!] =\!\boxed{\text{SP}}\!\Rightarrow E3$ (8)

$E3 \vdash [\![MAKE\ B\!: b.]\!] =\!\boxed{\text{SP}}\!\Rightarrow E4$ (9)

$E4 \vdash [\![JOIN\ ip\ \rightarrow\ b.]\!] =\!\boxed{\text{SP}}\!\Rightarrow E5$ (10)

$E5 \vdash [\![b]\!] =\!\boxed{\text{U}}\!\Rightarrow kunit : ktype_u\ ,\ E6$ (11)

$Check\text{-}Joins(E6)$ (12)

$Scope\text{-}End(E',\ E6)\ =\ E''$ (13)

$E \vdash [\![\textit{constset}\colon \textit{unit}]\!] =\!\boxed{\mathrm{CA}}\!\Rightarrow \textit{kconstset},\ \textit{kunit}\colon \textit{ktypeconst},\ \textit{ktype},\ E'$

(6) $: E1 = ($ [**typedec**(bool)],
[**fndec**(NOR)],
[**signaldec**(in1, **bool**, **input**), **signaldec**(in2, **bool**, **input**)],
{NOR→1}, {}
{bool→, f→, t→}, {}
{in2→, in1→}, {}
{}, {}, {}
)

(7) $: E1 \vdash [\![(in1, in2)]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow \textit{kunit} : \textit{ktype},\ E1'$ (14)
$E1 \vdash [\![\textit{LET ip} = (in1, in2).]\!] =\!\boxed{\mathrm{SP}}\!\Rightarrow$ *Add-Signal*(**signaldec**(*ip*, *ktype*, *kunit*), **joined**, $E1'$) (15)

(14) $: E1 \vdash [\![in1]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow \textit{kunit}_1 : \textit{ktype}_1,\ E11$ (16)
$E11 \vdash [\![in2]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow \textit{kunit}_2 : \textit{ktype}_2,\ E12$ (17)
$E1 \vdash [\![(in1, in2)]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow \textit{UnitTuple}([\textit{kunit}_1, \textit{kunit}_2]) :$
$\textit{TypeTuple}([\textit{ktype}_1, \textit{ktype}_2]),\ E12$ (18)

(16) : *Find-Signal*(*in*1, *E*1) ≡ let *Find-Sig-Nm*(*in*1, *E*1) = **sig**(1, **joined**) in
signal(1), **bool**
$E11 \equiv E1$
$E1 \vdash [\![in1]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow$ **signal**(1): **bool**, $E1$

(17) $: E1 \vdash [\![in2]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow$ **signal**(2): **bool**, $E1$

(18) $: E1 \vdash [\![(in1, in2)]\!] =\!\boxed{\mathrm{U}}\!\Rightarrow$
UnitTuple([**signal**(1), **signal**(2)]): *TypeTuple*([**bool**, **bool**]), $E1$
= **units**([**signal**(1), **signal**(2)]) : **types**([**bool**, **bool**]) , $E1$

(15) : *Add-Signal*(**signaldec**(*ip*, **types**([**bool**, **bool**]),
units([**signal**(1), **signal**(2)])), **joined**, $E1$)
= let *Len* = 2 in
let *SigName* = *ip* in
*E*2

$E2 = ($ [**typedec**(bool)],
[**fndec**(NOR)],
[**signaldec**(in1, **bool**, **input**), **signaldec**(in2, **bool**, **input**),
signaldec(ip, **types**([**bool**, **bool**]), **units**([**signal**(1), **signal**(2)]))
],
{NOR→1}, {}
{bool→, f→, t→}, {}
{in2→, in1→}, {ip→ **sig**(3, **joined**)}
{}, {}, {}
)

(8) $: \mathit{Scope\text{-}Fn\text{-}Begin}(E2) \;=\; E2'$ (19)
$E2' \vdash [\![FN\ B\ ...]\!] \;=\!\boxed{\mathrm{FD}}\!\Rightarrow\; E2''$ (20)
$\mathit{Scope\text{-}Fn\text{-}End}(E2,\ E2'') \;=\; E3$ (21)
$E2 \vdash [\![FN\ B\ ...]\!] \;=\!\boxed{\mathrm{SP}}\!\Rightarrow\; E3$ (22)

(19) $: E2' = ($ [**typedec**(bool)],
[**fndec**(NOR)],
[],
{NOR→1}, {}
{bool→, f→, t→}, {}
{}, {},
{}, {}, {}
)

(20) $: E2' \vdash [\![(bool\!: ip1,\ bool\!: ip2)]\!] \;=\!\boxed{\mathrm{IN}}\!\Rightarrow\; ktype_i,\ E2'1$ (23)
$E2'1 \vdash [\![bool]\!] \;=\!\boxed{\mathrm{T}}\!\Rightarrow\; ktype_o$ (24)
$E2'1 \vdash [\![NOR(ip1, ip2)]\!] \;=\!\boxed{\mathrm{U}}\!\Rightarrow\; kunit\!:\ ktype_u,\ E2'2$ (25)
$ktype_o \;=\!\boxed{\mathrm{T}}\!=\; ktype_u$ (26)
$E2' \vdash [\![FN\ B\ ...]\!] \;=\!\boxed{\mathrm{FD}}\!\Rightarrow\; \mathit{Add\text{-}Fn}(\ \mathbf{fndec}(B, ktype_i, E2'2.sigdec, ktype_o, kunit),$
$E2',\ E2'2)$ (27)

(23) $: ktype_i \;=\;$ [**bool**, **bool**]

$E2'1 = ($ [**typedec**(bool)],
[**fndec**(NOR)],
[**signaldec**(ip1, **bool**, **input**), **signaldec**(ip2, **bool**, **input**)],
{NOR→1}, {}
{bool→, f→, t→}, {}
{}, {ip1→ **sig**(1, **joined**), ip2→ **sig**(2, **joined**)},
{}, {}, {}
)

(24) $: ktype_o \;=\;$ **bool**

(25) $: E2'1 \vdash [\![(ip1, ip2)]\!] \;=\!\boxed{\mathrm{U}}\!\Rightarrow\; kunit\!:\ ktype,\ E2'1'$ (28)
$fnno \;=\; \mathit{Find\text{-}Fn}(NOR,\ E2'1)$ (29)
$(E2'1.fndec)[fnno].inputtype \;=\!\boxed{\mathrm{T}}\!=\; ktype$ (30)
$E2'1 \vdash [\![NOR(ip1, ip2)]\!] \;=\!\boxed{\mathrm{U}}\!\Rightarrow\;$ **instance**$(fnno, kunit)$:
$(E.fndec)[fnno].outputtype,\ E2'1'$ (31)

(28) $: E2'1 \vdash [\![(ip1, ip2)]\!] \;=\!\boxed{\mathrm{U}}\!\Rightarrow\;$ **units**([**signal**(1), **signal**(2)]): **types**([**bool**, **bool**]), $E2'1$

(29) $: fnno \;=\; 1$

(30) : **types**([**bool**, **bool**]) $=\!\boxed{\mathrm{T}}\!=\; ktype$ (= **types**([**bool**, **bool**]))

(31) : $E2'1 \vdash$ **[**$NOR(ip1, ip2)$**]** $=\!\boxed{\text{U}}\!\Rightarrow$ **instance**(1, **units**([**signal**(1), **signal**(2)])): **bool**, $E2'1$

(26) : $ktype_o$ (= **bool**) $=\!\boxed{\text{T}}\!=$ $ktype_u$ (= **bool**)

(27) : $E2' \vdash$ **[**$FN\ B$ ...**]** $=\!\boxed{\text{FD}}\!\Rightarrow$ *Add-Fn*(**fndec** (B,
types([**bool**, **bool**]),
[**signaldec**(ip1, **bool**, **input**),
signaldec(ip2, **bool**, **input**)],
bool,
instance(1, **units**([**signal**(1), **signal**(2)]))
),
$E2'$, $E2'1$)
$\equiv E2''$

$E2''$= ([**typedec**(bool)],
[**fndec**(NOR),
fndec(B,
types([**bool**, **bool**]),
[**signaldec**(ip1, **bool**, **input**), **signaldec**(ip2, **bool**, **input**)],
bool,
instance(1, **units**([**signal**(1), **signal**(2)]))
],
[],
{NOR→1}, {B→2}
{bool→, f→, t→}, {}
{}, {},
{}, {}, {}
)

(21) : *Scope-Fn-End*($E2$, $E2''$) = $E3$

$E3$ = ([**typedec**(bool)],
[**fndec**(NOR),
fndec(B,
types([**bool**, **bool**]),
[**signaldec**(ip1, **bool**, **input**), **signaldec**(ip2, **bool**, **input**)],
bool,
instance(1, **units**([**signal**(1), **signal**(2)]))
],
[**signaldec**(in1, **bool**, **input**), **signaldec**(in2, **bool**, **input**),
signaldec(ip, **types**([**bool**, **bool**]), **units**([**signal**(1), **signal**(2)]))
],
{NOR→1}, {B→2}
{bool→, f→, t→}, {}
{in2→, in1→}, {ip→ **sig**(3, **joined**)}
{}, {}, {}
)

(22) : $E2 \vdash$ **[**$FN\ B$ ...**]** $=\!\boxed{\text{SP}}\!\Rightarrow$ $E3$

$$(9)\quad : \textit{Find-Fn}(B,\ E3) = \textit{fnno} \qquad (32)$$
$$\textit{ktype}_i = E.\textit{fndec}[\textit{fnno}].\textit{inputtype},\ \textit{ktype}_o = E.\textit{fndec}[\textit{fnno}].\textit{outputtype} \qquad (33)$$
$$E \vdash [\textit{MAKE } B\text{: } b.] \boxed{\text{SP}}\!\!\Rightarrow \textit{Add-Signal}(\ \textbf{signaldec}(b, \textit{ktype}, \textbf{instance}(\textit{fnno}, \textbf{unitquery}(\textit{ktype})), \textbf{unjoined}, E3) \qquad (34)$$

$$(32)\quad : \textit{Find-Fn}(B,\ E3) = 2$$

$$(33)\quad : \textit{ktype}_i = \ \textbf{types}([\ \textbf{bool},\ \textbf{bool}]),\ \textit{ktype}_o = \ \textbf{bool}$$

$$(34)\quad : \textit{Add-Signal}(\ \textbf{signaldec}(b,\ \textbf{bool},\ \textbf{instance}(2,\ \textbf{unitquery}(\ \textbf{types}([\ \textbf{bool},\ \textbf{bool}])))),\ \textbf{unjoined}, E3) = E4$$

```
E4 = ( [ typedec(bool)],
       [ fndec(NOR),
         fndec(B,
               types([ bool, bool]),
              [ signaldec(ip1, bool, input), signaldec(ip2, bool, input)],
               bool,
               instance(1, units([ signal(1), signal(2)]))
       ],
       [ signaldec(in1, bool, input), signaldec(in2, bool, input),
         signaldec(ip, types([ bool, bool]), units([ signal(1), signal(2)])),
         signaldec(b, bool, instance(2, unitquery( types([ bool, bool]) )))
       ],
       {NOR→1}, {B→2}
       {bool→, f→, t→}, {}
       {in2→, in1→}, {ip→ sig(3, joined),
                       b → sig(4, unjoined) }
       {}, {}, {}
     )
```

$$(10)\quad : E4 \vdash [\textit{ip}] \boxed{\text{U}}\!\!\Rightarrow \textit{kunit} : \textit{ktype}\ ,\ E4' \qquad (35)$$
$$\textit{Find-Unjoined}(b, E4) = \textit{fnno} \qquad (36)$$
$$\textit{Find-Signal}(b, E4) = \ \textbf{signal}(\textit{signalno}) : \textit{ktype}_o \qquad (37)$$
$$E4.\textit{fndec}[\textit{fnno}].\textit{inputtype} \ \boxed{\text{T}}\!\!= \textit{ktype} \qquad (38)$$
$$E4 \vdash [\textit{JOIN ip} \rightarrow b.] \boxed{\text{SP}}\!\!\Rightarrow \textit{Add-Join}(\ \textbf{signaldec}(b, \textit{ktype}_o, \textbf{instance}(\textit{fnno}, \textit{kunit})), \textit{signalno}, E4') \qquad (39)$$

$$(35)\quad : \textit{Find-Signal}(\textit{ip}, E4) \equiv \text{let } \textit{Find-Sig-Nm}(\textit{ip}, E4) = \ \textbf{sig}(3,\ \textbf{joined}) \text{ in } \textbf{signal}(3),\ \textbf{types}([\ \textbf{bool},\ \textbf{bool}])$$

$$E4 \vdash [\textit{ip}] \boxed{\text{U}}\!\!\Rightarrow \ \textbf{signal}(3)\text{: } \ \textbf{types}([\ \textbf{bool},\ \textbf{bool}]),\ E4$$

(36) : *Find-Unjoined*$(b, E4)$ $\equiv$
let *Signo* = $(E4.lclsignamemap)(b).signalno \equiv 4$ in
let **signaldec**(b, **bool**, **instance**(*fnno*, **unitquery**(**types**([**bool**, **bool**])))) in
= $(E4.sigdec)[Signo]$
fnno $\equiv$ 2

(37) : *Find-Signal*$(b, E4)$ $\equiv$ let *Find-Sig-Nm*$(b, E4)$ = **sig**(4, **unjoined**) in
signal(4), **bool**

(38) : $(E4.fndec)[2].inputtype$ (= **types**([**bool**, **bool**])) $=\boxed{T}\models$ *ktype* (= **types**([**bool**, **bool**]))

(39) : *Add-Join*(**signaldec**(b, **bool**, **instance**(2, **signal**(3)), 4, $E4$) = $E5$

```
E5 = ( [ typedec(bool)],
       [ fndec(NOR),
         fndec(B,
                 types([ bool, bool]),
                [ signaldec(ip1, bool, input), signaldec(ip2, bool, input)],
                 bool,
                 instance(1, units([ signal(1), signal(2)])) )
       ],
       [ signaldec(in1, bool, input), signaldec(in2, bool, input),
         signaldec(ip, types([ bool, bool]), units([ signal(1), signal(2)])),
         signaldec(b, bool, instance(2, signal(3)))
       ],
       {NOR→1}, {B→2}
       {bool→, f→, t→}, {}
       {in2→, in1→}, {ip→ sig(3, joined),
                        b → sig(4, joined) }
       {}, {}, {}
     )
```

(11) : *Find-Signal*$(b, E5)$ $\equiv$ let *Find-Sig-Nm*$(b, E5)$ = **sig**(4, **joined**) in
signal(4), **bool**

$E5 \vdash [b] =\boxed{U}\Rightarrow$ **signal**(4): **bool**, $E6$

where

```
E6 = ( [ typedec(bool)],
       [ fndec(NOR),
         fndec(B,
               types([ bool, bool]),
              [ signaldec(ip1, bool, input),  signaldec(ip2, bool, input)],
               bool,
               instance(1, units([ signal(1), signal(2)])) )
       ],
       [ signaldec(in1, bool, input),  signaldec(in2, bool, input),
         signaldec(ip,  types([ bool, bool]),  units([ signal(1), signal(2)])),
         signaldec(b,  bool,  instance(2, signal(3)))
       ],
       {NOR→1}, {B→2}
       {bool→, f→, t→}, {}
       {in2→, in1→}, {ip→ sig(3,  joined),
                          b → sig(4,  joined) }
       {}, {}, {}
      )
```

(12) : *Check-Joins*($E6$) = *True*

(13) : *Scope-End*(E', $E6$) = E''

```
E'' = ( [ typedec(bool)],
        [ fndec(NOR),
          fndec(B,
                types([ bool, bool]),
               [ signaldec(ip1, bool, input),  signaldec(ip2, bool, input)],
                bool,
                instance(1, units([ signal(1), signal(2)])))
        ],
        [ signaldec(in1, bool, input),  signaldec(in2, bool, input),
          signaldec(ip,  types([ bool, bool]),  units([ signal(1), signal(2)])),
          signaldec(b,  bool,  instance(2, signal(3)))
        ],
        {}, {NOR → 1}
        {}, {bool →, f→, t→}
        {}, {in2→ sig(2,  joined), in1→ sig(1,  joined)}
        {}, {B}, {ip, b}
       )
```

(4) : $ktype_o$ (= **bool**) $=\boxed{T}=$ $ktype_u$ (= **bool**)

```
(5)   : Add-Fn( fndec( A,
                       types([ bool, bool]),
                      [ signaldec(in1, bool, input), signaldec(in2, bool, input),
                        signaldec(ip, types([ bool, bool]), units([ signal(1), signal(2)])),
                        signaldec(b, bool, instance(2, signal(3)))
                      ],
                       bool,
                       signal(4)
                      ),
              E, E''
              )
      ≡ E'''

      E''' = ( [ typedec(bool)],
               [ fndec(NOR),
                 fndec(B,
                       types([ bool, bool]),
                      [ signaldec(ip1, bool, input), signaldec(ip2, bool, input)],
                       bool,
                       instance(1, units([ signal(1), signal(2)])) )
                 fndec(A,
                       types([ bool, bool]),
                      [ signaldec(in1, bool, input), signaldec(in2, bool, input),
                        signaldec(ip, types([ bool, bool]), units([ signal(1), signal(2)])),
                        signaldec(b, bool, instance(2, signal(3)))
                      ],
                       bool,
                       signal(4) )
               ],
               [],
               {}, {NOR → 1, A → 3},
               {}, {bool→, f→, t→}
               {}, {}
               {}, {}, {}
             )
```

3.7.4 The Kernel Closure

The closure of the **Kernel** with declarations *bool*, *NOR* and *A* is defined in the following way

[CL] $$\begin{array}{l} : \mathcal{E}_1 \vdash [\![\mathit{TYPE\ bool\ =\ ...}]\!] \dashv\!\boxed{\mathrm{TD}}\!\Rightarrow \mathcal{E}_2 \\ \mathcal{E}_2 \vdash [\![\mathit{FN\ NOR\ =\ ...}]\!] \dashv\!\boxed{\mathrm{FD}}\!\Rightarrow \mathcal{E}_3 \\ \mathcal{E}_3 \vdash [\![\mathit{FN\ A\ =\ ...}]\!] \dashv\!\boxed{\mathrm{FD}}\!\Rightarrow \mathcal{E}_4 \\ \hline \mathcal{E}_1 \vdash [\![\mathit{closure}]\!] \dashv\!\boxed{\mathrm{CL}}\!\Rightarrow \mathcal{E}_4 \end{array}$$

[KERNEL] : $[\![\mathit{closure}]\!] \dashv\!\boxed{\mathrm{KERNEL}}\!\Rightarrow (\mathcal{E}_4.\mathit{typedec},\ \mathcal{E}_4.\mathit{fndec})$

where

$\mathcal{E}_1 = ([], [], [], \{\}, \{\}, \{\}, \{\}, \{\}, \{\}, \{\}, \{\}, \{\})$

$\mathcal{E}_4 = E'''$

and thus

```
E4.typedecs   =  [ typedec(bool, tags([ tag(t,{}), tag(f,{})])) ]

E4.fndecs     =  [ fndec(NOR,
                        types([ bool, bool]),
                        [ signaldec(in1, bool, input), signaldec(in2, bool, input)],
                        bool,
                        caseclause( units([ signal(1), signal(2)]),
                                  [ case( constsets([ enum(1,2), enum(1,1)]), enum(1,2)),
                                    case( constsets([ enum(1,1), enum(1,2)]), enum(1,2)),
                                    case( constsets([ enum(1,1), enum(1,1)]), enum(1,2))],
                                    enum(1,1)) )
                   fndec(B,
                        types([ bool, bool]),
                        [ signaldec(ip1, bool, input), signaldec(ip2, bool, input)],
                        bool,
                        instance(1, units([ signal(1), signal(2)])) )
                   fndec(A,
                        types([ bool, bool]),
                        [ signaldec(in1, bool, input), signaldec(in2, bool, input),
                          signaldec(ip, types([ bool, bool]), units([ signal(1), signal(2)])),
                          signaldec(b, bool, instance(2, signal(3)))
                        ],
                        bool,
                        signal(4) )
                 ]
```

4 Conclusions

In this document we have demonstrated how, by means of a set of software transformations and a set of formal transformations, any ELLA description can be mapped into a set of data structures. The software transformations are a suite of transformations which map ELLA descriptions onto its Core. The formal transformation system define a set of rules which map Core constructs onto a set of data structures known as the **Kernel**. The mapping onto the **Kernel**, together with the use of a specific transformation environment, provide a formal definition of the static semantics of the Core. Examples of both transformation processes have been given.

5 Acknowledgements

The authors would like to acknowledge the support of the IED project 4/1/1357 "Formal Verification Support for ELLA". The authors would also like to thank E.V. Whiting of DRA/ED (RSRE) and H. Barringer, G. Gough, B. Monahan of Manchester University Computer Science Department for their many constructive comments on this work.

A Glossary of Symbols

Functions

$f: D_1 \times D_2 \rightarrow R$	signature
$f \underline{\Delta} \cdots$	function definition
$f(d)$	application
if $\cdots$ *then* $\cdots$ *else* $\cdots$	condtional
let $x = \cdots$ *in* $\cdots$	local definition
case x *of* $\cdots$ *else* $\cdots$ *end*	choice
pre	pre-condition

Composite Objects

Object: : *fieldname*: *fieldtype*	Record Object definition
$\mu(E, s \mapsto t)$	change field s of E to hold t
$\mu(E, s \mapsto (E.s \dagger t))$	update field s of E by overwriting with t

Sets

T-set	finite subset of T
$\{t_1, \cdots, t_k\}$	set enumeration
$\{\}$	empty set
$t \in T$	set membership
$T_1 \cap T_2$	set intersection
$T_1 \cup T_2$	set union
$T_1 \subseteq T_2$	set containment
$\mathbb{Z}$	$\{\cdots$, -1, 0, 1, $\cdots\}$
$\mathbb{N}_1$	$\{1, 2, \cdots\}$
$\mathbb{B}$	{true, false}

Maps

$D \xrightarrow{m} R$	finite map
dom m	domain
rng m	range
$m_1 \dagger m_2$	overwriting

Sequences

S^*	finite sequence
$[s_1, \cdots, s_k]$	sequence enumeration
$[\,]$	empty sequence
len l	length of sequence l
$s_1 \frown s_2$	concaternation
$\iota(i \in \text{inds}\ sequence) \cdot sequence[i] = s$	The unique element of *sequence* which equals s

Transformation

$E =$ **env**(_,_,_,_,_,_,_,_,_,_,_,_)	Transformation Environment
E.*fieldname*	field selection in the **Kernel**
(E.*filedname*)[*number*]	indexing
inv(E)	invariant of environment E
Environment $\vdash$ [*Core-Syntax*] $=\boxed{\text{Rule}}\Rightarrow$ *Kernel-Expressions*	formal transformation
$=\boxed{\text{U}}\Rightarrow$ _:_,_	syntactic separaters (: ,)
$ktype_1 =\boxed{\text{T}}= ktype_2$	type equality in the **Kernel**

Kernel

typedec(_,_)	**Kernel** data structure with wild-card entries
TypeOpt	Type structure with optional element nil
TypeSeq	*Non-empty sequence of Types*
kType	'Type' belonging in the **Kernel**

B ELLA Composite Syntax

This appendix presents the ELLA V6 Syntax[1]

B.1 Basic Notation

abc $\in$ Abc	'abc' is an element of the set 'Abc'
b ::= c	the syntax definition of 'b' is 'c'
|	the separator of alternatives in a syntax definition
|	ELLA separator of alternatives
intorstring	an ELLA integer name or string of printable characters
$d_1 \cdots d_k$	one or more occurrences of 'd'
$d_1, \cdots, d_k$	one or more occurrences of 'd' separated by ','. Note if k=1 then no ',' is present.
$d_1, \cdots, d_{k-1}$	zero or more occurrences of 'd' separated by ','. Note if k=0 then no ',' is present.
$z \in \mathbb{Z}$	$z \in \{ \cdots, -1, 0, 1, \cdots \}$
$j,k \in \mathbb{N}_1$	$j,k \in \{ 1, 2, \cdots \}$
Identifier	Lower case letter
Fnname	Upper case letter or symbol
Integer	ELLA integer expression
Constant	ELLA constant expression
Character	Any printable character

B.2 Syntactic Categories

typename	$\in$	Identifier	(ELLA type name i.e. lower case)
intname	$\in$	Identifier	(Integer name i.e. lower case)
constname	$\in$	Identifier	(ELLA constant name i.e. lower case)
signalname	$\in$	Identifier	(ELLA signal name)
tagname	$\in$	Identifier	(ELLA tagged type name)
altname	$\in$	Identifier	(ELLA enumerated type alternative)
macintname	$\in$	Identifier	(Macro integer parameter name)
mactypename	$\in$	Identifier	(Macro type parameter name)
repname	$\in$	Identifier	(Replicator variable name)
nullname	$\in$	Identifier	(An unnamed identifier, given by a blank space)
attributename	$\in$	Identifier	(An attribute identifier name)
contextname	$\in$	Identifier	(a context name)
subregionname	$\in$	Identifier	(a context subregion name)
fnname	$\in$	Fnname	(ELLA function name i.e. upper case or symbol)
biopname	$\in$	Fnname	(ELLA BIOP name)
alienname	$\in$	Fnname	(ELLA ALIEN name)
index	$\in$	Integer	(index of structure)
lwb, upb	$\in$	Integer	(Lower-bound and Upper-bound of a range)
size	$\in$	Integer	(Size of row or string)
interval	$\in$	Integer	(ELLA timing interval)

ambigtime	∈	Integer	(Ambiguity delay time)
delaytime	∈	Integer	(delay time)
skewtime	∈	Integer	(skew time value of input data)
slowtime	∈	Integer	(slow down factor)
fasttime	∈	Integer	(speed up factor)
initialvalue	∈	Constant	(Delay, Retiming or Ram initialisation value)
ambigvalue	∈	Constant	(Delay ambiguity value)
char	∈	Character	(A printable character i.e. 'a')
firstchar	∈	Character	(First printable character in a range)
lastchar	∈	Character	(Last printable character in a range)
string	∈	String	(A string of printable characters i.e. 'abc')

B.3 Syntactic Definitions

Enumerated

```
enumerated    ::=   altname
                  | tagname / integer
                  | tagname 'char
                  | tagname "string"
```

Integer

```
integer       ::=   integer dop integer1
                  | integer1

integer1      ::=   integer1 + integer2
                  | integer1 - integer2
                  | integer2

integer2      ::=   integer2 * integer3
                  | integer2 % integer3
                  | integer3

integer3      ::=   z
                  | intname
                  | signalname
                  | mop integer3
                  | IF bool THEN integer ELSE integer FI
                  | ( integer )

dop           ::=   SL
                  | SR
                  | IAND
                  | IOR
                  | MOD
```

mop	::=		+
		|	-
		|	INOT
		|	ABS
		|	SQRT

Bool

bool	::=		integer bop1 integer
		|	bool bop2 bool
		|	NOT bool
bop1	::=		=
		|	/ =
		|	>
		|	<
		|	>=
		|	<=
bop2	::=		AND
		|	OR

Type

type	::=		type → type
		|	type1
type1	::=		()
		|	typename
		|	STRING [size] typename
		|	[size] type1
		|	(type_1 , $\cdots$, type_k)

Constant

const	::=		STRING [size] const1
		|	[size] const
		|	const1
const1	::=		constname
		|	enumerated
		|	altname & const1
		|	tagname ('firstchar .. 'lastchar)
		|	tagname / (lwb .. upb)
		|	($\mathrm{constset}_1$, $\cdots$, $\mathrm{constset}_k$)
		|	? type1
		|	type1

Constset

constset	::=	const_1 |$\cdots$ |const_k

Unit

unit	::=	unit CONC unit1
	|	unit fnname fnparams attributes unit1
	|	unit1
unit1	::=	STRING [size] unit1
	|	[size] unit1
	|	[INT repname = lwb .. upb] unit1
	|	fnname fnparams attributes unit1
	|	altname & unit1
	|	unit2 // altname
	|	unit2 attributes
unit2	::=	signalname
	|	enumerated
	|	IO signalname
	|	unit2 [index]
	|	unit2 [lwb .. upb]
	|	unit2 [[unit]]
	|	REPLACE (unit, unit, unit)
	|	IF bool THEN unit ELSE unit FI
	|	? type1
	|	closedclause
fnparams	::=	{ params_1 , $\cdots$, params_k }
	|	nullname
params	::=	integer
	|	type
	|	const
	|	fnname
	|	macname fnparams
attributes	::=	$\text{attname}_1 \cdots \text{attname}_{k-1}$
attname	::=	@ attributename

Closedclause

closedclause	::=	**CASE unit OF cases elsecases ELSE unit ESAC**
	|	**CASE unit OF cases elsecases ESAC**
	|	**()**
	|	**(unit_1 , $\cdots$, unit_k)**
	|	**BEGIN step_1 $\cdots$ step_{k-1} OUTPUT unit END**
	|	**BEGIN step_1 $\cdots$ step_k END**
	|	**(step_1 $\cdots$ step_{k-1} OUTPUT unit)**
	|	**(step_1 $\cdots$ step_k)**
	|	**BEGIN SEQ sequence_1 ; $\cdots$; sequence_{k-1} ; OUTPUT unit END**
	|	**BEGIN SEQ sequence_1 ; $\cdots$; sequence_{k-1} END**
	|	**(SEQ sequence_1 ; $\cdots$; sequence_{k-1} ; OUTPUT unit)**
	|	**(SEQ sequence_1 ; $\cdots$; sequence_{k-1})**
cases	::=	constset_1 : unit_1 , $\cdots$, constset_k : unit_k
elsecases	::=	elseof_1 $\cdots$ elseof_{k-1}
elseof	::=	ELSEOF cases
step	::=	declarations .
	|	LET letdecs_1 , $\cdots$, letdecs_k .
	|	MAKE makedecs_1 , $\cdots$, makedecs_k .
	|	JOIN joindecs_1 , $\cdots$, joindecs_k .
	|	FOR multiplejoin_1 $\cdots$ multiplejoin_k JOIN joindecs_1 , $\cdots$, joindecs_j .
	|	PRINT printitem_1 , $\cdots$, printitem_k .
	|	FAULT printitem_1 , $\cdots$, printitem_k .
declarations	::=	INT intdec_1 , $\cdots$, intdec_k
	|	TYPE typedec_1 , $\cdots$, typedec_k
	|	CONST constdec_1 , $\cdots$, constdec_k
	|	FN fndec_1 , $\cdots$, fndec_k
	|	MAC macdec_1 , $\cdots$, macdec_k
decnames	::=	signalname
	|	(sigornullname_1 , $\cdots$, sigornullname_k)
letdecs	::=	decnames = unit
makedecs	::=	**fnname fnparams fnparams attributes : signalname_1 $\cdots$ signalname_k**
	|	**[size] makedecs**
joindecs	::=	**unit $\rightarrow$ joinunit**
joinunit	::=	**joinunit CONC joinunit1**
	|	**joinunit1**

joinunit1 ::= [INT repname = lwb .. upb] joinunit1
| joinunit2

joinunit2 ::= signalname
| IO signalname
| joinunit2 [lwb .. upb]
| joinunit2 [index]
| (joinunit$_1$, $\cdots$, joinunit$_k$)

multiplejoin ::= INT repname = lwb .. upb

printitem ::= IF bool THEN intorstring$_1$ $\cdots$ intorstring$_k$ FI
| intorstring$_1$ $\cdots$ intorstring$_k$

sequence ::= declarations
| LET letdecs$_1$, $\cdots$, letdecs$_k$
| VAR vardecs$_1$, $\cdots$, vardecs$_k$
| PVAR pvardecs$_1$, $\cdots$, pvardecs$_k$
| PRINT printitem$_1$, $\cdots$, printitem$_k$
| FAULT printitem$_1$, $\cdots$, printitem$_k$
| sequence2

sequence2 ::= assign := unit
| (multassign$_1$, $\cdots$, multassign$_k$) := unit
| CASE unit OF seqcase seqelseof ELSE sequence2 ESAC
| CASE unit OF seqcase seqelseof ESAC
| IF bool THEN sequence2 ELSE sequence2 FI
| IF bool THEN sequence2 FI
| [INT repname = lwb .. upb] sequence2
| (sequence2$_1$; $\cdots$; sequence2$_k$)

seqcase ::= seqstep$_1$, $\cdots$, seqstep$_k$

seqstep ::= constset : sequence2
| constset :

seqelsecase ::= seqelseof$_1$ $\cdots$ seqelseof$_{k-1}$

seqelseof ::= ELSEOF seqcase

vardecs ::= decnames := unit

pvardecs ::= decnames ::= const

assign ::= signalname
| assign [lwb .. upb]
| assign [index]
| assign [[unit]]

multassign	::=	assign
	|	nullname

Function Body

functionbody	::=	unit
	|	**REFORM**
	|	**BIOP** biopname fnparams
	|	**ALIEN** alienname fnparams
	|	**ARITH** integer
	|	**DELAY** delaybody
	|	**IDELAY** (initialvalue , delaytime)
	|	**SAMPLE** samplebody
	|	**FASTER** fasterbody
	|	**SLOWER** slowerbody
	|	**RAM** (initialvalue)
	|	**IMPORT**

delaybody	::=	(initialvalue , ambigtime , ambigvalue , delaytime)
	|	(initialvalue , ambigtime , delaytime)
	|	(initialvalue , delaytime)

samplebody	::=	(interval , initialvalue , skewtime)
	|	(interval)

fasterbody	::=	(fnname , fasttime , initialvalue , skewtime)
	|	(fnname , fasttime)

slowerbody	::=	(fnname , slowtime , initialvalue , skewtime)
	|	(fnname , slowtime)

Integer Declaration

intdec	::=	intname = integer

Type Declaration

typedec	::=	typename = typeornew

typeornew	::=	type
	|	**NEW** tagname / (lwb .. upb)
	|	**NEW** (typealt_1 |··· |typealt_k)
	|	**NEW** tagname (charange_1 |··· |charange_k)

typealt	::=	altname & typel
	|	altname

charange	::=	'char
	|	'firstchar .. 'lastchar

Constant Declaration

constdec	::=	constname = constset

Function Declaration

fndec	::=	fnname = input → outtype : functionbody
input	::=	terminals
	|	()
terminals	::=	(terminaltype_1 , $\cdots$, terminaltype_k)
terminaltype	::=	type : $\text{sigornullname}_1 \cdots \text{sigornullname}_k$
	|	type
sigornullname	::=	signalname
	|	nullname
outtype	::=	terminals
	|	type

Macro Declaration

macdec	::=	fnname maclist = macinput → outtype : functionbody
maclist	::=	{ macpram_1 , $\cdots$, macpram_k }
	|	nullname
macpram	::=	INT $\text{intname}_1 \cdots \text{intname}_k$
	|	TYPE $\text{typename}_1 \cdots \text{typename}_k$
	|	CONST mactype : $\text{constname}_1 \cdots \text{constname}_k$
	|	FN (mactype) → mactype : $\text{fnname}_1 \cdots \text{fnname}_k$
	|	FN () → mactype : $\text{fnname}_1 \cdots \text{fnname}_k$
	|	MAC maclist (mactype) → type: $\text{fnname}_1 \cdots \text{fnname}_k$
	|	MAC maclist () → type: $\text{fnname}_1 \cdots \text{fnname}_k$
macinput	::=	macterminals
	|	()
macterminals	::=	(mactermtype_1, $\cdots$, mactermtype_k)

mactermtype	::=	mactype : $\text{sigornullname}_1 \cdots \text{sigornullname}_k$
	|	mactype
mactype	::=	mactype $\rightarrow$ mactype
	|	mactype1
mactype1	::=	()
	|	typename
	|	STRING [size] typename
	|	[size] mactype1
	|	($\text{mactype}_1 , \cdots , \text{mactype}_k$)
	|	mactype2
mactype2	::=	[INT macintname] mactype
	|	STRING [INT macintname] typename
	|	TYPE mactypename

Imports

imports	::=	IMPORTS importgroup , $\cdots$, importgroup .
importgroup	::=	context : importfn $\cdots$ importfn
importfn	::=	fnname
	|	fnname (RENAMED fnname)
	|	fnname RENAMED fnname
context	::=	contextname
	|	contextname / subregionname

Closure

closure	::=	declarations $\cdots$ declarations
	|	declarations $\cdots$ declarations imports

C Core ELLA Composite Syntax

C.1 Basic Notation

abc ∈ Abc	'abc' is an element of the set 'Abc'
b ::= c	the syntax definition of 'b' is 'c'
|	the separator of alternatives in a syntax definition
|	ELLA separator of alternatives
$d_1 \cdots d_k$	one or more occurrences of 'd'
$d_1, \cdots, d_k$	one or more occurrences of 'd' separated by ','. Note if k=1 then no ',' is present.
$d_1, \cdots, d_{k-1}$	zero or more occurrences of 'd' separated by ','. Note if k=0 then no ',' is present.
$\mathbb{Z}$	$\{ \cdots, -1, 0, 1, \cdots \}$
$\mathbb{N}_1$	$\{ 1, 2, \cdots \}$
Identifier	Lower case letter
Fnname	Upper case letter or symbol
Constant	ELLA constant expression
Character	Any printable character

C.2 Syntactic Categories

typename	∈	Identifier	(ELLA type name e.g. lower case)
signalname	∈	Identifier	(ELLA signal name)
tagname	∈	Identifier	(ELLA tagged type name)
altname	∈	Identifier	(ELLA enumerated type alternative)
fnname	∈	Fnname	(ELLA function name e.g. upper case or symbol)
biopname	∈	Fnname	(ELLA BIOP name e.g. upper case)
z	∈	$\mathbb{Z}$	(An integer)
lwb, upb	∈	$\mathbb{Z}$	(An integer)
j,k	∈	$\mathbb{N}_1$	(A non-zero positive integer)
index	∈	$\mathbb{N}_1$	(A non-zero positive integer)
size	∈	$\mathbb{N}_1$	(A non-zero positive integer)
interval	∈	$\mathbb{N}_1$	(ELLA timing interval)
ambigtime	∈	$\mathbb{N}_1$	(Ambiguity delay time)
delaytime	∈	$\mathbb{N}_1$	(delay time)
skewtime	∈	$\mathbb{N}_1$	(skew delay)
initialvalue	∈	Constant	(Delay, Retiming or Ram initialisation value)
ambigvalue	∈	Constant	(Delay ambiguity value)
char	∈	Character	(A printable character e.g. 'a')
string	∈	String	(A string of printable characters e.g. 'abc')

C.3 Syntactic Definitions

Enumerated

```
enumerated  ::=   altname
                | tagname / z
                | tagname 'char
                | tagname "string"
```

Type

```
type        ::=   typename
                | STRING [ size ] typename
                | [ size ] type
                | ( type1, ···, typek )
                | ()
```

Constant

```
const       ::=   STRING [ size ] const1
                | [ size ] const
                | const1

const1      ::=   enumerated
                | altname & const1
                | ( const1, ···, constk )
                | ? type
                | ()
```

Constset

```
constset    ::=   constset1 |··· |constsetk

constset1   ::=   STRING [ size ] constset2
                | [ size ] constset1
                | constset2

constset2   ::=   enumerated
                | altname & constset2
                | ( constset1, ···, constsetk )
                | type
```

Unit

unit	::=	unit CONC unit1
	|	unit1
unit1	::=	STRING [size] unit1
	|	[size] unit1
	|	fnname unit1
	|	altname & unit1
	|	unit2 // altname
	|	unit2
unit2	::=	signalname
	|	enumerated
	|	unit2 [index]
	|	unit2 [index_{lwb} .. index_{upb}]
	|	unit2 [[unit]]
	|	REPLACE (unit, unit, unit)
	|	? type
	|	closedclause

Closedclause

closedclause	::=	CASE unit OF cases ELSE unit ESAC
	|	($\text{unit}_1, \cdots, \text{unit}_k$)
	|	BEGIN $\text{step}_1 \cdots \text{step}_{k-1}$ OUTPUT unit END
	|	()
cases	::=	$\text{constset}_1 : \text{unit}_1, \cdots, \text{constset}_k : \text{unit}_k$
step	::=	typedec
	|	fndec
	|	LET signalname = unit .
	|	MAKE fnname : signalname .
	|	JOIN unit → signalname .

Function Body

functionbody	::=	unit
	|	REFORM
	|	BIOP biopname
	|	DELAY (initialvalue, ambigtime, ambigvalue, delaytime)
	|	IDELAY (initialvalue, delaytime)
	|	SAMPLE (interval, initialvalue, skewtime)
	|	RAM (initialvalue)

Type Declaration

typedec	::=	TYPE **typename** = **typeornew**.
typeornew	::=	**type**
	|	**new**
new	::=	NEW **tagname** / (**lwb** .. **upb**)
	|	NEW ($\mathbf{typealt}_1$ | ⋯ | $\mathbf{typealt}_k$)
	|	NEW **tagname** ('$\mathbf{char}_1$ |⋯ |'$\mathbf{char}_k$)
typealt	::=	**altname** & **type**
	|	**altname**

Function Declaration

fndec	::=	FN fnname = input → type : functionbody.
input	::=	(type_1 : signalname_1, ⋯, type_k : signalname_k)
	|	()

Closure

declaration	::=	typedec
	|	fndec
closure	::=	$\text{declaration}_1 \cdots \text{declaration}_k$

D Kernel of ELLA Data Structure

D.1 Conventions

abc	$\in$	Abc (ie. it is an element of the set Abc)
Indexer, Size, Fnno	$\subseteq$	$\mathbb{N}_1$
Typeno, Tagno, Inputno	$\subseteq$	$\mathbb{N}_1$
Signalno, Ambigtime, Delaytime	$\subseteq$	$\mathbb{N}_1$
Interval, Skew	$\subseteq$	$\mathbb{N}_1$
Inputtype, Outputtype	$\subseteq$	Type
Initialvalue, Ambigvalue	$\subseteq$	Const
Fnname, Biopname	$\subseteq$	Upper case identifier or operator
Name, Signalname	$\subseteq$	Lower case identifier
Typename, Tagname	$\subseteq$	Lower case identifier
Lowerbound, Upperbound	$\subseteq$	positive or negative integer
Character	$\subseteq$	printable character

D.2 Kernel Data Structure

Enumerated

```
Enumerated  ::=   Enum
                | string( Typeno × TagnoSeq )

Enum        ::=   enum( Typeno × Tagno )
```

Types

```
Type        ::=   typeno( Typeno )
                | typename( Typename × Type )
                | stringtype( Size × Type )
                | types( TypeSeq )
                | typevoid
```

Constants

```
Const       ::=   Enumerated
                | conststring( Size × Const )
                | consts( ConstSeq )
                | constassoc( Enum × Const )
                | constquery( Type )
                | constvoid
```

Constant Sets

Constset	::=	Enumerated
	|	**constsetalts**(ConstsetSeq)
	|	**constsetstring**(Size × Constset)
	|	**constsets**(ConstsetSeq)
	|	**constsetassoc**(Enum × Constset)
	|	**constsetany**(Type)

Units

Unit	::=	Enumerated
	|	**conc**(Unit × Unit × Outputtype)
	|	**unitstring**(Size × Unit)
	|	**units**(UnitSeq)
	|	**instance**(Fnno × Unit)
	|	**unitassoc**(Enum × Unit)
	|	**extract**(Unit × Enum)
	|	**signal**(Signalno)
	|	**index**(Unit × Indexer × Outputtype)
	|	**trim**(Unit × Indexer × Indexer × Outputtype)
	|	**dyindex**(Unit × Unit × Outputtype)
	|	**replace**(Unit × Unit × Unit)
	|	**unitquery**(Type)
	|	**caseclause**(Unit × CaseSeq × Unit)
	|	**unitvoid**

Case	::=	**case**(Constset × Unit)

Function Declarations

Fndec	::=	**fndec**(Fnname × Inputtype × SignaldecSeq × Outputtype × Fnbody)
Signaldec	::=	**signaldec**(Signalname × Type × Unitorinput)
Unitorinput	::=	Unit
	|	**input**
Fnbody	::=	Unit
	|	**reform**
	|	**biop**(Biopname)
	|	**delay**(Initialvalue × Ambigtime × Ambigvalue × Delaytime)
	|	**idelay**(Initialvalue × Delaytime)
	|	**sample**(Interval × Initialvalue × Skew)
	|	**ram**(Initialvalue)

Type Declarations

Typedec	::=	**typedec**(Typename × New)
New	::=	**tags**(TagSeq)
	|	**ellaint**(Tagname × Lowerbound × Upperbound)
	|	**chars**(Tagname × CharacterSeq)
Tag	::=	**tag**(Tagname × TypeOpt)

Closures

Closure	::=	TypedecSeq × FndecSeq

E FIFO Example

This appendix presents a description of a fifo written in high level ELLA together with an automatically transformed version of the circuit.

E.1 High Level Description

```
                    # Type Declarations  #

TYPE bool = NEW ( t | f | x ),
     int  = NEW i/(0..100).


                # Fifo - data goes into the first empty cell. #
                # ----                                        #

MAC FIFO {INT size} = (int:data_in, bool:shift_in, bool:shift_out) -> (int, bool):
(SEQ
   PVAR fifo ::= [size](i/0, f);                  # create state variable  #

   CASE shift_out
   OF  t : fifo := fifo[2..size] CONC (i/0, f)    # remove element         #
   ESAC;

   VAR entered := f;
   CASE shift_in                                  # add element            #
   OF  t : [INT i = 1..size-1]
              CASE (entered, fifo[i][2])
              OF  (f,f) : ( fifo[i] := (data_in, t);
                                entered := t
                          )
              ESAC
   ESAC;

   OUTPUT (fifo[1][1], entered)
).

FN FIFO_9 = (int: data_in, bool: shift_in, bool: shift_out) -> (int, bool):
FIFO {9} (data_in, shift_in, shift_out).                        # call macro #
```

E.2 Transformed Description

This section presents the result of applying the built-in software transformations of ELLA to the above FIFO circuit. Note that names which begin with an 's' followed by a number are automatically generated by the transformation process. Also function names of the form 'NAME # {..} #' are instantiations of macros where the items between the hash comments are the parameters which have be used in the instantiation.

```
TYPE bool = NEW( t | f | x ).

TYPE int = NEW i/( 0..100 ).

FN F1_DELAY #{[ 9 ]( int, bool ), [ 9 ]( i/0, f )}# = ( [ 9 ]( int, bool )) ->
                                                       [ 9 ]( int, bool ):
   DELAY( [ 9 ]( i/0, f ), 1 ).

FN FIFO #{9}# = (int: data_in, bool: shift_in, bool: shift_out) -> (int, bool):
   ( MAKE F1_DELAY #{[ 9 ]( int, bool ), [ 9 ]( i/0, f )}# : s6fifo.
     LET s7fifo =
         CASE shift_out
         OF t: ( LET s8fifo =
                     s6fifo[ 2..9 ] CONC ( i/0, f ).
                 OUTPUT s8fifo
               )
         ELSE s6fifo
         ESAC.
     LET s9entered = f.
     LET ( fifo, entered ) =
          CASE shift_in
          OF t: ( LET ( s12fifo, s13entered ) =
                        CASE ( s9entered,s7fifo[1 ][ 2 ] )
                        OF ( f,f ):( LET s14fifo = ( data_in,t ) CONC
                                                           s7fifo[2..9 ].
                                     LET s15entered = t.
                                     OUTPUT ( s14fifo, s15entered )
                                    )
                        ELSE ( s7fifo, s9entered )
                        ESAC.
                  LET ( s16fifo, s17entered ) =
                        CASE ( s13entered, s12fifo[2 ][ 2 ] )
                        OF ( f, f ): ( LET s18fifo = ( s12fifo[1 ] CONC
                                                     ( data_in, t )) CONC
                                                           s12fifo[3..9 ].
                                       LET s19entered = t.
                                       OUTPUT ( s18fifo, s19entered )
                                     )
                        ELSE ( s12fifo, s13entered )
                        ESAC.
                  LET ( s20fifo, s21entered ) =
                        CASE ( s17entered, s16fifo[3 ][ 2 ] )
                        OF ( f, f ): ( LET s22fifo = ( s16fifo[1..2 ] CONC
                                                     ( data_in, t )) CONC
                                                           s16fifo[4..9 ].
                                       LET s23entered = t.
                                       OUTPUT ( s22fifo, s23entered )
                                     )
                        ELSE ( s16fifo, s17entered )
                        ESAC.
```

```
LET ( s24fifo, s25entered ) =
      CASE ( s21entered, s20fifo[4 ][ 2 ] )
      OF ( f, f ): ( LET s26fifo = ( s20fifo[1..3 ] CONC
                                     ( data_in, t )) CONC
                                           s20fifo[5..9 ].
                     LET s27entered = t.
                     OUTPUT ( s26fifo, s27entered )
                   )
      ELSE ( s20fifo, s21entered )
      ESAC.
LET ( s28fifo, s29entered ) =
      CASE ( s25entered, s24fifo[5 ][ 2 ] )
      OF ( f, f ): ( LET s30fifo = ( s24fifo[1..4 ] CONC
                                     ( data_in, t )) CONC
                                           s24fifo[6..9 ].
                     LET s31entered = t.
                     OUTPUT ( s30fifo, s31entered )
                   )
      ELSE ( s24fifo, s25entered )
      ESAC.
LET ( s32fifo, s33entered ) =
      CASE ( s29entered, s28fifo[6 ][ 2 ] )
      OF ( f, f ): ( LET s34fifo = ( s28fifo[1..5 ] CONC
                                     ( data_in, t )) CONC
                                           s28fifo[7..9 ].
                     LET s35entered = t.
                     OUTPUT ( s34fifo, s35entered )
                   )
      ELSE ( s28fifo, s29entered )
      ESAC.
LET ( s36fifo, s37entered ) =
      CASE ( s33entered, s32fifo[7 ][ 2 ] )
      OF ( f, f ): ( LET s38fifo = ( s32fifo[1..6 ] CONC
                                     ( data_in, t )) CONC
                                           s32fifo[8..9 ].
                     LET s39entered = t.
                     OUTPUT ( s38fifo, s39entered )
                   )
      ELSE ( s32fifo, s33entered )
      ESAC.
LET ( s40fifo, s41entered ) =
      CASE ( s37entered, s36fifo[8 ][ 2 ] )
      OF ( f, f ): ( LET s42fifo = ( s36fifo[1..7 ] CONC
                                     ( data_in, t )) CONC
                                              s36fifo[9 ].
                     LET s43entered = t.
                     OUTPUT ( s42fifo, s43entered )
                   )
      ELSE ( s36fifo, s37entered )
      ESAC.
```

```
                  OUTPUT ( s40fifo, s41entered )
                )
        ELSE ( s7fifo, s9entered )
        ESAC.
   JOIN fifo -> s6fifo.
   OUTPUT ( fifo[ 1 ][ 1 ], entered )
  ).

FN FIFO_9 = ( int: data_in, bool: shift_in, bool: shift_out ) -> ( int, bool ):
  FIFO #{9}# ( data_in, shift_in, shift_out ).
```

It can be noted that the transformed function is a valid description in Core ELLA.

F Three Pump Controller

F.1 Introduction

This appendix presents a high level, medium level and low level description in ELLA of a three pump controller. The definition of the controller is given in [Bar91] and is reproduced here

A reservoir is connected to a lake by a pipe line. Water is taken from the lake to the reservoir by a system of three pumps.
Three level sensors are installed on the reservoir. Their outputs are denoted by signals a_1, a_2, a_3. Signal a_i is 0 when the water is above level i, for $i = 1,2,3$ and has a value 1 when the water is below level i. The number of pumps that are on at any one time depends on the water level in the reservoir. In particular: if the water level is between level 1 and 2, then one pump should be in operation; if the water level is between level 2 and 3, then two pumps should be in operation; if the water level is below level 3, then three pumps should be in operation. Of course, if the water level is above level 1 then no pumps should be in operation. In order to equalise wear on the pumps, they should come into operation in a cyclic manner.

F.2 High Level Description

In this section we give a high level description of the pump controller.

```
TYPE  pump = NEW (none | a | b | ab | c | ca | bc | abc ),
      level = NEW 1/(0..3),
      bool = NEW (t | f).

FN CONTROL = (level:in) -> pump:
( SEQ
      PVAR store ::= (none,t);
      store := CASE in OF
                  1/0 : (store[1],t),
                  1/1 : CASE store[1] OF
                              a | ca : (b,f),
                              b | ab : (c,f)
                              ELSE     (a,f)
                        ESAC,
                  1/2 : CASE store[1] OF
                              a | ab : (bc,f),
                              b | bc : (ca,f)
                              ELSE     (ab,f)
                        ESAC,
                  1/3 : (abc,f)
                ESAC;
      OUTPUT CASE store[2] OF
               t: none,
               f: store[1]
             ESAC
).
```

Three enumerated types have been defined. The first 'pump' denotes which pumps are actually operating, the pumps being known as 'a', 'b' and 'c'. At first glance the ordering of the enumcrated type might appear strange. However the ordering was chosen such that when the circuit is transformed to gate level the output of the controller will be a three bit signal, with each bit representing one of the pumps. The second type 'level' denotes the level of water in the reservoir, with zero representing a full reservoir. The third type is a boolean flag which is used in the monitoring of the active pump. The function CONTROL is the pump controller and its CASE clause sets up which pumps get switched on.

Although CONTROL has been written using sequences this is not really necessary. A functional version of CONTROL is therefore given, this being an equivalent description to the sequential form.

```
FN F1_DELAY = (( pump, bool )) -> ( pump, bool ): DELAY(( none, t ), 1 ).

FN CONTROL = ( level: in ) -> pump:
BEGIN
   MAKE F1_DELAY: s3store.
   LET store =
      CASE in OF
      1/0: ( s3store[ 1 ], t ),
      1/1:
         CASE s3store[ 1 ] OF
         a | ca: ( b, f ),
         b | ab: ( c, f )
         ELSE ( a, f )
         ESAC,
      1/2:
         CASE s3store[ 1 ] OF
         a | ab: ( bc, f ),
         b | bc: ( ca, f )
         ELSE ( ab, f )
         ESAC,
      1/3: ( abc, f )
      ESAC.
   JOIN store -> s3store.
   OUTPUT
      CASE store[ 2 ] OF
       t: none,
       f: store[ 1 ]
      ESAC
END.
```

F.3 Medium Level Description

This section presents the results of replacing the enumerated types for the pump switch's and level indicators by rows of two valued types. This synthesising of the types makes explicit the algorithm behind the type naming of the high level version. It would have been possible to describe the controller from the medium level from the outset, however the higher level version provides extra checks. In particular in the high level version the level indicators can only take

four possible values whereas in this medium level version they can take eight. This medium level version treats such illegal values as 'unknown' and causes the simulator to return the ELLA unknown value, whereas the high level version would explicitly indicate if the level integer range was violated.

This medium level version has maintained close correspondence with the high level version by the use of 'constant' statements. Thus the majority of the controller description has remained unaltered, hence reducing the likelihood of error. The complete description is given by

```
TYPE  switch = NEW (on | off),
      pump   = [3]switch,
      level  = [3]switch,
      bool   = NEW (t | f).

CONST none = (off, off, off),
      a    = (on, off, off),
      b    = (off, on, off),
      c    = (off, off, on),
      ab   = (on, on, off),
      ca   = (on, off, on),
      bc   = (off, on, on),
      abc  = (on, on, on).

CONST level0 = (off, off, off),
      level1 = (on, off, off),
      level2 = (on, on, off),
      level3 = (on, on, on).


FN CONTROL = (level:in) -> pump:
( SEQ
      PVAR store ::= (none,t);
      store := CASE in OF
                 level0 : (store[1],t),
                 level1 : CASE store[1] OF
                            a | ca : (b,f),
                            b | ab : (c,f)
                            ELSE     (a,f)
                          ESAC,
                 level2 : CASE store[1] OF
                            a | ab : (bc,f),
                            b | bc : (ca,f)
                            ELSE     (ab,f)
                          ESAC,
                 level3 : (abc,f)
               ESAC;
      OUTPUT CASE store[2] OF
               t: none,
               f: store[1]
             ESAC
).
```

F.4 Low Level Description

This section presents the results of synthesising the medium level description of the pump controller through the ELLA-GATEMAP [Pitt88] system. Apart from the functions F1_DELAY and CONTROL all the other functions are basic cells in one of the technology libraries that GATEMAP supports.

```
#-------------------------------------------------------------------------#
#                                                                         #
# ELLA netlist generated by GATEMAP II version 1.2                        #
#                                                                         #
# Module     : CONTROL                                                    #
# Date       : 16-MAY-1991 13:41                                          #
# Library    : USR$WORK:[]                                                #
# Technology : USR$GATEMAPROOT:[12.TECHNOLOGIES]*******                   #
#                                                                         #
#-------------------------------------------------------------------------#


#------------------------------ TYPES ---------------------------------#

TYPE bool = NEW( f | t | x | z ).

TYPE tech_bool = bool.

CONST logic_0 = f.


#------------------------- LIBRARY CELLS ----------------------------#

FN INV1 = ( bool: a ) -> bool: # Inverter #.

FN NAND2 = ( bool: a b ) -> bool: # Two Input NAND #

FN NAND3 = ( bool: a b c ) -> bool: # Three Input NAND #

FN NOR2 = ( bool: a b ) -> bool: # Two Input NOR #

FN NOR3 = ( bool: a b c ) -> bool: # Three Input NOR #

FN X2ANOR = ( bool: a b c d ) -> bool: NOR(AND(a,b), AND(c,d)).

FN EXNOR = ( bool: a b ) -> bool: # Two Input Exclusive OR #

FN CLKB = ( bool: ai ) -> ( bool, bool ): # Clock Drvier #

FN DF = ( bool: ckt cki d ) -> ( bool, bool ):  # Clocked Cell #
```

```
#---------------------- ELLA DELAY FUNCTION ---------------------------#

FN F1_DELAY = ( tech_bool: unnamed_input_1, tech_bool: unnamed_input_2,
                tech_bool: unnamed_input_3, tech_bool: unnamed_input_4 ) ->
              ( tech_bool, tech_bool, tech_bool, tech_bool ):
   BEGIN
       MAKE DF  : xcmp17 xcmp19 xcmp15 xcmp21,
            CLKB: xcmp18.
       JOIN ( xcmp18[ 1 ], xcmp18[ 2 ], unnamed_input_3 ) -> xcmp17,
            ( xcmp18[ 1 ], xcmp18[ 2 ], unnamed_input_2 ) -> xcmp19,
            ( logic_0 ) -> xcmp18,
            ( xcmp18[ 1 ], xcmp18[ 2 ], unnamed_input_1 ) -> xcmp15,
            ( xcmp18[ 1 ], xcmp18[ 2 ], unnamed_input_4 ) -> xcmp21.
       OUTPUT ( xcmp15[ 1 ], xcmp19[ 1 ], xcmp17[ 1 ], xcmp21[ 1 ] )
   END.

#------------------------- PUMP CONTROLLER ----------------------------#

FN CONTROL = ( tech_bool: in_1, tech_bool: in_2, tech_bool: in_3 ) ->
             ( tech_bool, tech_bool, tech_bool ):
   BEGIN
       MAKE INV1     : xcmp39 xcmp69 xcmp76 xcmp24 xcmp74 xcmp33 xcmp61 xcmp37,
            NAND2    : xcmp56 xcmp66 xcmp25 xcmp36 xcmp38 xcmp53 xcmp70
                       xcmp47 xcmp64,
            NAND3    : xcmp45 xcmp67 xcmp84 xcmp35,
            NOR3     : xcmp75 xcmp40 xcmp80,
            X2ANOR   : xcmp78 xcmp72 xcmp82,
            EXNOR    : xcmp48,
            F1_DELAY: xcmp4.
       JOIN ( xcmp72, xcmp67 ) -> xcmp56,
            ( xcmp37, in_2, xcmp47 ) -> xcmp45,
            ( xcmp56 ) -> xcmp39,
            ( xcmp76, xcmp4[ 1 ], xcmp74 ) -> xcmp75,
            ( xcmp37, in_2, xcmp66 ) -> xcmp35,
            ( in_1, xcmp4[ 2 ], xcmp37, xcmp80 ) -> xcmp78,
            ( xcmp61, xcmp4[ 2 ] ) -> xcmp66,
            ( in_1, xcmp4[ 1 ], xcmp37, xcmp75 ) -> xcmp72,
            ( xcmp53 ) -> xcmp69,
            ( in_2, xcmp4[ 1 ], xcmp4[ 3 ] ) -> xcmp40,
            ( in_3 ) -> xcmp76,
            ( xcmp24, xcmp37 ) -> xcmp25,
            ( in_1, xcmp4[ 3 ], in_3, xcmp48 ) -> xcmp82,
            ( xcmp78, xcmp35 ) -> xcmp36,
            ( xcmp37, in_2, xcmp64 ) -> xcmp67,
            ( xcmp39, xcmp37 ) -> xcmp38,
            ( xcmp37, xcmp4[ 1 ], xcmp4[ 2 ] ) -> xcmp84,
            ( xcmp36 ) -> xcmp24,
            ( xcmp4[ 3 ] ) -> xcmp74,
            ( xcmp4[ 2 ] ) -> xcmp33,
            ( xcmp4[ 1 ] ) -> xcmp61,
```

```
        ( xcmp82, xcmp45 ) -> xcmp53,
        ( in_1 ) -> xcmp37,
        ( xcmp69, xcmp37 ) -> xcmp70,
        ( xcmp33, xcmp4[ 3 ] ) -> xcmp47,
        ( xcmp66, xcmp47 ) -> xcmp64,
        ( xcmp76, xcmp61, xcmp4[ 2 ] ) -> xcmp80,
        ( xcmp40, xcmp84 ) -> xcmp48,
        ( xcmp56, xcmp36, xcmp53, xcmp37 ) -> xcmp4.
    OUTPUT ( xcmp38, xcmp25, xcmp70 )
END.
```

References

[All89] L. Allison. A Practical Introduction To Denotational Semantics. Cambridge Computer Science Texts 23, 1989

[BGHT90] R Boulton, M Gordon, J Herbert, and J Van Tassel. The HOL Verification of ELLA Designs. Technical report, University of Cambridge Computer Laboratory, 1990.

[BGM91] H. Barringer, G. Gough, and B. Monahan. Operational Semantics of Hardware Design Languages. Technical Report UMCS-91-2-2, University of Manchester Computer Science Department, February 1991.

[BHM90] H. Barringer, M. Hill, and B. Monahan. Towards an Operational Semantics of Ella. Technical Report 1.1a, Formal Verification Support for Ella, IED project 4/1/1357, October 1990.

[Bar91] H. Barringer. Private Communication. 1991

[BGL+91] H.Barringer, G.Gough, T.Longshaw, B.Monahan, M.Peim, A.Williams Semantics and Verification for Boolean Kernel ELLA using IO Automata. CHARME Workshop, Turin, June 1991

[Com90a] Computer General Electronic Design, The New Church, Henry Street, Bath, Avon, BA1 1JR, United Kingdom. *The ELLA Language Reference Manual.* 4.0th edition, 1990.

[Com90b] Computer General Electronic Design, The New Church, Henry Street, Bath, Avon, BA1 1JR, United Kingdom. *The ELLA User Manual* . 4.0th edition, 1990.

[Com90c] Computer General Electronic Design, The New Church, Henry Street, Bath, Avon, BA1 1JR, United Kingdom. *The ELLA Tutorial* . 4.0th edition, 1990.

[Com90d] Computer General Electronic Design, The New Church, Henry Street, Bath, Avon, BA1 1JR, United Kingdom. *The ELLANET Reference Manual* . 4.0th edition, 1990.

[Jon90] C.B. Jones. Systematic Software Development Using VDM. Prentice Hall, 1990

[HM91] M.G. Hill and J.D. Morision. Preliminary Core ELLA Definition. Technical report 1.1c, Formal Verification Support for Ella, IED project 4/1/1357, February 1991.

[HPC+90] M.G. Hill, N.E. Peeling, I.F. Currie, J.D. Morison, E.V. Whiting, and C.O. Newton. Real number arithmetic for mixed behavioural and structural descriptions. Proceedings of the IEE, 137(G):446–450, 1990.

[HWM90a] M.G. Hill, E.V. Whiting, and J.D. Morison. Formal Semantic Definition of ELLA Timing. Internal Memorandum 4436, Royal Signals and Radar Establishment, Great Malvern, 1990.

[HWM90b] M.G. Hill, E.V. Whiting, and J.D. Morison. SPRITE-ELLA language enhancements. Internal Memorandum 4441, Royal Signals and Radar Establishment, Great Malvern, 1990.

[HWM91] M.G. Hill, E.V. Whiting, and J.D. Morison. Bidirectionality, Connectivity and Instantiations in ELLA. Internal Memorandum 4421, Royal Signals and Radar Establishment, Great Malvern, 1991.

[Pitt88] E.B. Pitty. A Critique of the GATEMAP Logic Synthesis System. International Workshop on Logic and Architecture Synthesis For Silicon Compilers, Grenoble, May 1988

[Mon91] B.Q. Monahan. A note on the semantics of ambiguity. Project working paper, University of Manchester Computer Science Department, January 1991.

[MPT85] J.D.Morison, N.E.Peeling, T.L.Thorp The Design Rationale of ELLA, A Hardware Design and Description Language. CHDL 1985, Tokyo

[MPW87] J.D.Morison, N.E.Peeling, E.V.Whiting Sequential Programming Extension to ELLA, with Automatic Transformation to Structure. ICCD 1987, New York

[SWH88] D.J. Snell, E.V. Whiting, and M.G. Hill. The New Assembler. Technical Report, Royal Signals and Radar Establishment, Great Malvern, June 1988.

[Tai88a] S. Tait. System specification for a new simulator. Internal Memorandum P209.40.4, Praxis, 1988.

[Tai88b] S. Tait. Formal Specification of Built in Operations. Praxis, internal memo number P029.40.5, 1988

[DCT] D.C.Taylour An Overview of Recent ELLA Language Developments. To appear as RSRE Memorandum 4447

[WMW+89] J.S.Ward, J.D.Morison, E.V.Whiting, N.E.Peeling, M.G.Hill New Developments in ELLA. European Simulation Multi Conference, June 1989, Rome

REPORT DOCUMENTATION PAGE

DRIC Reference Number (if known) ....

Overall security classification of sheet ....................................UNCLASSIFIED....................................

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the field concerned must be marked to indicate the classification eg (R), (C) or (S).

Originators Reference/Report No.	Month	Year
REPORT 91024	AUGUST	1991

Originators Name and Location

RSRE, St Andrews Road
Malvern, Worcs WR14 3PS

Monitoring Agency Name and Location

Title

A FORMAL DEFINITION OF THE STATIC SEMANTICS OF ELLA'S CORE

Report Security Classification	Title Classification (U, R, C or S)
UNCLASSIFIED	U

Foreign Language Title (in the case of translations)

Conference Details

Agency Reference	Contract Number and Period

Project Number	Other References

Authors	Pagination and Ref
MORISON, J D; HILL, M G	102

Abstract

At the heart of the full ELLA language are a set of Core constructs into which any ELLA description can be transformed. This document describes a set of formal transformation rules which map these Core constructs into a set of data structures. These transformation rules define the static semantics of the language. Examples are given of circuits which are translated from the full language into ELLA's Core and of Core circuits which are translated via the formal transformation system into a set of data structures.

Abstract Classification (U,R,C or S)

U

Descriptors

Distribution Statement (Enter any limitations on the distribution of the document)

UNLIMITED

S80/48